


UNIVERSITY OF ALBERTA LIBRARY



0 0004 1973 322



EX LIBRIS
UNIVERSITATIS
ALBERTÆNSIS



Digitized by the Internet Archive
in 2023 with funding from
University of Alberta Library

<https://archive.org/details/Garraway1993>

UNIVERSITY OF ALBERTA
RELEASE FORM

NAME OF AUTHOR: Robert William Thomas Garraway

TITLE OF THESIS: Hierarchical Control and Management in a CAI Visual
Authoring Environment

DEGREE: Doctor of Philosophy

YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

UNIVERSITY OF ALBERTA

Hierarchical Control and Management
in a CAI Visual Authoring Environment

BY

Robert William Thomas Garraway



A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

DEPARTMENT OF EDUCATIONAL PSYCHOLOGY

Edmonton, Alberta

SPRING 1993

UNIVERSITY OF ALBERTA

Department of Chemistry and Physics
in the Faculty of Science

14



100 Street, Edmonton, Alberta

A book containing the following chapters:
1. Introduction to the study of chemistry
2. The structure of matter
3. The properties of matter
4. The laws of chemistry
5. The history of chemistry

UNIVERSITY OF ALBERTA

100 Street, Edmonton, Alberta

100 Street, Edmonton, Alberta

THE UNIVERSITY OF ALBERTA
FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled Hierarchical Control and Management in a CAI Visual Authoring Environment submitted by Robert William Thomas Garraway in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

Dedication

To my wife, Bahiya, and my children, Naisan and Yasmin, and
to the graduates, students, and staff of Maxwell International Bahá'í School.

Abstract

A review of the historical development of CAI from the *creative breakthrough* in 1958, through the *replication* period [1959-66], the *empirical* period [1967-74], the *theoretical* period [1975-82], and the *automation* period [1983-90] was carried out. The evolution of the sequence control and courseware management aspects of CAI languages and authoring systems was examined. These two aspects are the focus of this thesis. It was found that almost all CAI languages and authoring systems tend to provide a two level system of management and control: a within-file system and a separate and distinct between-file system. It was concluded that a more unified multi-level system of management and control features in a CAI system would enhance courseware organization, design and development. Ideas for the design of a large scale CAI system were also contributed from the following areas of computer science: the concept of abstraction, visual programming, human-computer interaction, and graphical user interfaces.

The design of a large scale, multi-user CAI system was proposed based on a modular CAI language, ERAS, which has six sub-languages: CONTROL, CONTENT, DISPLAY, INPUT, ANSWER, and MENU. The system supported a hierarchical courseware data base and a visual authoring environment. It was designed to have a unified look and feel for all classes of users, to incorporate features that support user and courseware registration, and to assist authors at the design stage of courseware development.

Two single user prototypes were developed to test some of the design features and the user interface, one on a Digital Equipment VAX using Elf (Educational Language Facility) and the other on a Commodore AMIGA using Intuition and GFABasic with Extend. The features tested included a multi-branch tree structured data base, the nodes of which held all courseware

modules. The authoring environment was maintained on an execution stack as the author navigated the tree structure. The visual authoring environment consisted of a file card metaphor of the data in the tree structure as well as a tree structure editor which presented contextual information on the tree node currently being edited.

Acknowledgements

I would particularly like to thank my supervisor, Dr. Steve Hunka, for his guidance, patience, and encouragement throughout the duration of this research. I also greatly appreciate the comments and assistance given by my committee members. I will warmly remember working with the community of researchers that made up the Division of Educational Research Services, in particular Alan Davis, Jim Hunka, Jamie Higham, William Chiu, Norm McGinnis, and John Nesbit.

Finally, I wish to thank my children and my wife for their patience and long suffering.

Table of Contents

Chapter	Page
1. Introduction and Historical Perspective	1
1.1 CAI and the Future of Education	1
1.2 The Development of Computer Systems	2
1.3 The Development of Computer-Assisted Instruction	6
1.3.1 Replication period [1959-66]	6
1.3.2 Empirical period [1967-74]	8
1.3.3 Theoretical Period [1975-82]	12
1.3.4 Automation period [1983-90]	16
1.4 Aspects of Computer-Assisted Instruction	17
1.5 Future Needs	18
2. The Problem and Expectations	21
2.1 The Problem	21
2.1.1 Evolution of Sequence Control and Courseware Management	22
2.1.2 Second Generation Authoring Languages	23
2.1.3 Third Generation Authoring Languages	32
2.1.4 Fourth Generation Authoring Systems	40
2.1.5 Beyond Lessonware	56
2.1.6 Hierarchical Control and Management	58
2.1.7 Authoring Environments	59

2.2	Expectations	60
2.2.1	The ERAS Authoring Language	61
2.2.2	The Hierarchical Courseware Database	62
2.2.3	The Visual Authoring Environment	66
3.	Contributing Ideas	70
3.1	The Concept of Abstraction	70
3.2	Visual Programming	74
3.3	Human-Computer Interaction	80
3.4	Graphical User Interfaces	85
4.	System Design	90
4.1	Desirable Characteristics	90
4.2	The System Environment	92
4.3	Courseware Management	95
4.3.1	LEVELs and ENTRY LEVELs	95
4.3.2	Identifiers	98
4.3.3	Scope of Identifiers	101
4.3.4	Case Sensitivity of Identifiers	102
4.3.5	Restart Points and Control Modes	102
4.4	Sequence Control	104
4.5	Computation	108
4.5.1	Variables and Constants	109
4.6	Visual Authoring	109
4.6.1	File Card Metaphor	111
4.6.2	Tree Structure Editor	113
4.6.3	Other Editors	114

5. Prototype Implementations	117
5.1 The Elf System and the Commodore AMIGA	117
5.2 Courseware Storage	118
5.3 The Run-time System	120
5.4 The Editors	122
5.5 The Authoring Environment	123
5.6 The Elf Interface	124
5.7 The Elf File card Metaphor	126
5.8 The AMIGA Interface	131
5.9 The Menus and the List Selector	134
5.9.1 The Pull Down Menus	134
5.9.2 The List Selector	138
5.10 The File card Editor	141
5.11 The Tree Structure Editor	155
5.12 The Systems Editor	160
6. Conclusions and Assessment	165
6.1 Expectations Attained	165
6.2 Conclusions	170
6.3 Assessment	171
6.4 Future Research	172
BIBLIOGRAPHY	174
Appendix A Elf - The Educational Language Facility	186
Appendix B ERAS Types, Variables, Constants, and Expressions	195

Appendix C ERAS Control Language	206
Appendix D ERAS Content Language	217
Appendix E ERAS Answer Language	222

List of Tables

Table	Page
1. Computer Generations and Underlying Basic Technologies	4
2. Types of Directories Supported	65
3. Pull Down Menus	137

List of Figures

Figure	Page
1. LEVEL Directories	64
2. Courseware Components	93
3. LEVELs and ENTRY LEVELs	97
4. Example of Generic Level Names	99
5. Example Course Tree	100
6. Module Invocation	106
7. Visual Authoring Environment	119
8. Execution Stack	121
9. New "Course" Structures	135
10. Tree Display Stack Frames and Label Lists	158

List of Computer Screens

Screen	Page
1. Elf Media File card	127
2. Internal Documentation Editor	128
3. Generic Level File cards	129
4. LEVEL File card	130
5. ERAS System File card	133
6. Course Selection	142
7. ENTRY LEVEL File Card	143
8. Generic Names Requester	145
9. Generic Names List	146
10. Generic Name Creation	148
11. LEVEL File Card	149
12. Next Level Selector	151
13. Cascade of LEVEL File Cards	153
14. "Exit to" Menu	154
15. Tree Structure Editor	156
16. "Courses" Window	161
17. ENTRY LEVEL Window	162
18. LEVEL Window	164

1. Introduction and Historical Perspective

The National Task Force on Educational Technology presented its report, titled "Transforming American Education: Reducing the Risk to the Nation", to the US Secretary of Education in the spring of 1986. The Task Force (cited in Technology News, 1986) commented:

To transform education, we must create a system in which an individual learning plan permits each learner to proceed at a rate and pace that is challenging but achievable, makes no unjust comparisons with the progress of others, prevents students from becoming passive, and assures positive reinforcement and steady progress. Such a plan will allow the most able to move to new realms without restriction and the least able to find their own unique achievement levels. (p. 4)

1.1 CAI and the Future of Education

It seems obvious that computer technology must be utilized if such far reaching educational objectives are to be achieved. Experimental use of computers in instruction began over thirty years ago (Rath, Anderson, & Brainerd, 1959) and has been progressing ever since. However, computer-assisted instruction (CAI) is mainly used as an adjunct to the regular school programs and has not been fully exploited to achieve the individualization of instruction as envisaged in the Task Force's Report.

Both hardware and software have been limiting factors. Although the advent of the microcomputer in 1977 made the large scale entry of computing power into the classroom a reality, the integration of these systems into large networks, seen as a necessary condition if computers are to make a direct impact on the organization of instruction, has yet to be realized. Also, a great increase in the quantity, the quality and the variety of courseware must be achieved. This has been understood by proponents of CAI for some time.

Further advancements in computer-based education depend in great measure on the creation, validation and maintenance of high-quality curriculum materials. If quality materials are to be produced, the techniques for developing those materials and describing them for computer systems must

not detract from attention to the substance and procedures of instruction. If persons expert in teaching and research in the disciplines (and it is the leaders in each field who should be doing the authoring) are to adopt computer-based education procedures, the authoring systems must be commensurate with their expertise and standards of quality. (Zinn, 1974, p. 381)

Providing tools for CAI authors who are not computer programmers has been a growing concern of instructional computer systems designers since the pioneering work of Romaniuk (1970) and Paloian (1971). As will be shown later, other computer users have also been concerned with similar issues in other areas of computer use.

This thesis is concerned with utilizing research from the fields of computer programming theory, visual programming, and human-computer interaction in the design and development of those tools needed by CAI authors in the sequence control and courseware management aspects of large scale CAI projects. (See section 1.4 for the definitions of these aspects of CAI.)

1.2 The Development of Computer Systems

It is quite common for introductory textbooks on educational computing to describe the history of computing since the Second World War as being divided into three or four generations that are distinguished by advances in electronics. For example, Lockard, Abrams, and Many (1987) describe the first generation as beginning in 1951 with the release of the first commercial computer based on vacuum tube technology, the second generation as beginning in the mid-1950s with the advent of the transistor, the third generation as beginning in the early 1960s with the advance to the integrated circuit, and finally the fourth generation as beginning in the early 1970s with the introduction of microelectronics.

Gaines and Shaw (1986b), however, have given a much broader and theoretical "analysis of the pattern of development of computer technology" (p. 21). They view computer technology as being composed of seven underlying basic technologies.

Successive breakthroughs in each of these technologies has initiated a new generation in overall computer technology. They see these generations as having an eight year period. The computer generation number, the year of the breakthrough that began the generation, and the name of the underlying basic technology within which the breakthrough occurred are given in Table 1. Each of the underlying technologies follows a cycle of development. Each period of the cycle occurs during one of the generations of computer technology. Gaines and Shaw (1986b) have termed these periods as follows:

Breakthrough:	creative advance made
Replication period:	experience gained by mimicking breakthrough
Empirical period:	design rules formulated from experience
Theoretical period:	underlying theories formulated and tested
Automation period:	theories predict experience & generate rules
Maturity:	theories become assimilated and used routinely (p. 7)

For example, at the end of the first generation [1955] of computer technology the breakthroughs in problem-oriented languages "triggered off" the second generation [1956-1963]. Thus the replication period for problem-oriented languages was during the second generation. This was when compilers were developed for FORTRAN, ALGOL, and COBOL. The empirical period was in the third generation [1964-1971], the theoretical period during the fourth generation [1972-1979], the automation period in the fifth generation [1980-1987], and finally the maturity period in the sixth generation [1988-].

It should be noted that this theoretical framework is somewhat different than the common reference to the growth of computer languages as progressing through four generations. In this scheme (Martin, 1983) the first generation is machine

Table 1. Computer Generations and Underlying Basic Technologies

Generation	Commencing	Underlying Basic Technology
0	1940	Electronic Device Technology
1	1948	Virtual Machine Architecture
2	1956	Problem-Oriented Languages
3	1964	Human-Computer Interaction
4	1972	Knowledge-Based Systems
5	1980	Inductive Inference Systems
6	1988	Autonomous Activity Systems

language; the second generation is symbolic assembly language; the third generation is high level languages like ALGOL, FORTRAN, COBOL, BASIC, Pascal, and Ada; and the fourth generation is high productivity languages like spreadsheets, database management systems, word processors, and computer-aided design systems. The rise of a wide variety of fourth generation languages can be viewed as the need to provide the non-professional programmer with the tools required to develop useful computer applications in a wide variety of fields.

Some of the characteristics of the generations of computer technology as given by Gaines and Shaw (1986b) are summarized below:

- 0th Generation [1940-47] - relays to vacuum tubes, designer as user, ENIAC, COLOSSUS (Good, 1980; Randell, 1980)
- 1st Generation [1948-55] - tubes, delay lines, drums, EDSAC, EDVAC, UNIVAC, IBM 701, 702, 650, WHIRLWIND, numeric control, person adapts to machine
- 2nd Generation [1956-63] - transistors and core stores, I/O control programs, IBM 704, 7090, 1401, PDP 1, 3, 4, 5, FORTRAN, ALGOL, COBOL, batch, execs, supervisors, console ergonomics, job control languages, simulators, graphics, BASIC, LISP 1.5
- 3rd Generation [1964-71] - large-scale integrated circuits, interactive terminals, IBM 360, 370, CDC 6600, 7600, PDP 6, 7, 8, 9, 10, DBMS, relational model, Intel 1103, 4004, time-sharing services, speech synthesis, APL/360, unix, shell, ELIZA
- 4th Generation [1972-79] - personal computers, supercomputers, VLSI, very large file stores, databanks, videotext, IBM 370/168 - MOS memory and virtual memory, DEC VAX, Intel 8080, dialogue rules, Altair and Apple PCs, Visicalc, PROLOG, Smalltalk, MYCIN
- 5th Generation [1980-87] - PCs with power and storage of mainframes plus graphics and speech processing, networks, utilities, NAPLPS standard, IBM 370 chip, HP-9000 chip with 450,000 transistors, Xerox star, IBM PC, Apple Macintosh, Videodisc, LISP and PROLOG machines, expert system shells, fifth generation project, knowledge bases
- 6th Generation [1988-93] - (speculative) optical logic and storage, organic processor elements, AI in routine use, integrated multi-modal systems, emotion detection, parallel knowledge systems, sixth generation project (p. 9)

1.3 The Development of Computer-Assisted Instruction

Stemming from a challenge from Prof. W. McGill, then at Columbia University, to some members of the IBM Research Laboratories, the thought of using a general purpose digital computer to teach was probably first conceived. (Rath, 1967b, p. 60)

This premier application of computers to direct instruction was first reported by Rath, Anderson, and Brainerd (1959) and took place in the IBM Research Center in Ossining, N.Y. in 1958. They simulated a teaching machine on an IBM 650 computer with a typewriter console to teach binary arithmetic. This was the birth place of what has become known today as computer-assisted instruction.

This "breakthrough" occurred during the second generation of the computer era. If the progress of CAI followed a similar pattern and metric to that discussed above for the underlying basic technologies of computing then the periods for the development of CAI might be predicted as follows:

- 1959-66 Replication period
- 1967-74 Empirical period
- 1975-82 Theoretical period
- 1983-90 Automation period
- 1990- Maturity period

In reviewing the historical literature it will be seen that the development of CAI does indeed follow very closely to this predicted scale.

1.3.1 Replication period [1959-66]

During this period most of the early experiments in CAI began. The first steps at bringing CAI into schools and universities were undertaken. And the first computer languages especially designed for writing CAI courseware were created. Listed below are some of the major activities of this period:

1959 - Bolt Beranek and Newman, Inc. (BBN) used an LGP-30 computer to teach foreign language vocabulary. (Rath, 1967b)

1960 - At the IBM Mohansic Laboratories an IBM 650 was used to develop a time-sharing CAI system with six terminals, audio random access memory, and a 1000 slide projector. To program courseware IBM created the Coursewriter language. CAI lessons in stenotyping, German, and statistics were produced. (Rath, 1967b)

Summer 1960 - The PLATO (Programmed Logic for Automatic Teaching Operation) project commenced at the University of Illinois, Urbana as one terminal connected to the ILLIAC I computer teaching high school math. (Rath, 1967b)

October 1960 - PLATO II was launched as a two terminal time-sharing system on ILLIAC I and later on a CDC 1604. (Bitzer, Hicks, Johnson, & Lyman, 1967; Rath, 1967b)

Early 1960s - Systems Development Corporation (SDC) began a CAI project using a Bendix G-15 computer. By 1966 they had developed the second specialized computer language for CAI, PLANIT (Programming LAnguage for Interactive Teaching). (Feingold & Frye, 1966; Rath, 1967b; Romaniuk, 1970)

- Pennsylvania State University began its research and development program in CAI. (Alessi & Trollip, 1985)

- Florida State University offered credit courses in physics and statistics via CAI. (Lockard et al., 1987)

January 1963 - A program of research and development in CAI began under Patrick Suppes and Richard Atkinson at Stanford University. Their first system evolved around a DEC PDP-1 computer with six student stations. Specialized multimedia devices were developed for the project by various manufacturers: a one second random-access optical display device for 512 pages of microfilm with a light pen by IBM (a predecessor of the IBM 1500), a 120 character vector graphic 1024 x 1024 CRT display and keyboard by Philco Corporation, and a one second random-access variable length audio play back unit by Westinghouse. (*Brief History*, 1968)

February 1963 - The Training Research Laboratory at the University of Illinois ordered the first computer to be used solely for research in computer based instruction (CBI). (Romaniuk, 1970)

October 1964 - IBM announced the first commercial "package" CBI system based on the Coursewriter I language. (Romaniuk, 1970)

1964 - EDUCOM was set up for the sharing and exchange of computer resources with members of the system. (Kearsley, Hunter, & Seidal, 1983b)

- PLATO III with 20 student stations was developed on the CDC 1604. Each student station had a special keyset and a video screen with access to a central bank of 122 slides. The keyset had specially labeled function keys that helped control the logic of the CAI lessons. (This became a feature of all PLATO systems to the present day.) Courseware was programmed with a modified FORTRAN-60 compiler. Work on the high resolution graphic plasma student terminal was under way. (Bitzer et al., 1967; Hickey, 1968)

1964-66 - The Stanford group began CAI experiments in various schools by means of teletypes connected to their computer via commercial telephone lines. (*Brief History*, 1968)

The capstone for the Replication period was the publication of a special issue of *IEEE Transactions on Human Factors in Electronics* in June 1967 dedicated to the research and development taking place in computer-assisted instruction. The guest editor was Gustave Rath who had been involved in the "Breakthrough" at IBM in 1958.

1.3.2 Empirical period [1967-74]

This period might be characterized as the time of the development of large scale CAI projects and powerful CAI languages, and the spread of CAI research beyond the borders of the United States. Also a great deal of data was collected on the efficacy and efficiency of CAI. Near the beginning of this period, Hickey (1968) listed 36

school districts, universities, and industrial and military centers where some type of CAI work was being done. In 1966 he had documentation for 140 CAI programs, but within two years this had more than doubled to 310. Some highlights from this period are listed below:

1967 - The period began, appropriately, with the introduction of the IBM 1500

Instructional System, the first integrated system specifically designed for computer based instruction. It supported a maximum of 32 multimedia student learning stations, each with a CRT terminal, lightpen and keyboard, a random-access 1024 frame 16mm film projector (the audio track being used for frame addressing), and a random-access variable length playback and record audio unit. The system could be programmed with both the Coursewriter II authoring language and a version of APL called MAT (Mathematical Algorithm Translator). By September 1967, systems had been delivered to Stanford University for the Brentwood School project, the University of Texas, Florida State University, the State University of New York (SUNY) at Stonybrook, and the Naval Academy. (Hickey, 1968; Kearsley et al., 1983a; Romaniuk, 1970)

- Suppes' group at Stanford founded the Computer Curriculum Corporation (CCC). They had refined an adaptive model of drill and practice and produced the "strands" curriculum in math, reading, and language arts. This was delivered by a Data General mini-computer with 96 CRT terminals. (Garraway, 1983; Kearsley et al., 1983b)

- Suppes' group also set up an IBM 1500 laboratory in the Brentwood Elementary School, East Palo Alto, California, and continued research in providing drill and practice CAI via teletype terminals in thirty schools in California, Iowa, Kentucky, and Mississippi. Dial-a-drill with touch tone phones in homes was also attempted. (*Brief History*, 1968; Hickey, 1968)

- Paul Tenczar, while working on his PhD in zoology, got tired of programming the PLATO III system in FORTRAN and so defined and helped implement the TUTOR CAI language. It had about 70 commands and became the standard programming language on both PLATO III and, later, PLATO IV. (Avner & Tenczar, 1969)

1968 - One of the first CAI projects outside of the United States was started under the direction of Steve Hunka at the Division of Educational Research Services of the University of Alberta, Edmonton, Canada with the arrival of an IBM 1500 system. This system was the last 1500 to remain in service and closed down in April 1980. (Garraway, 1983)

- The PILOT (Programmed Inquiry, Learning, Or Training) CAI language was defined by Starkweather at the University of California, San Francisco. It had eight single letter commands and was the first attempt to create a very simple yet powerful language for non-programmers to use to define courseware. (Barker, 1987; Lockard et al., 1987)

- The regular publication of indexes to courseware began with ENTELEK. (Kearsley et al., 1983b)

1969 - Alfred Bork began his work at the University of California, Irvine on instructing undergraduate physics using CAI with graphics and simulations. (Chambers & Sprecher, 1983)

Late 1960s - CAI projects began in the United Kingdom at the University of London's Queen Mary and Chelsea Colleges, Leeds University, and the artificial intelligence laboratory of Edinburgh University. (Chambers & Sprecher, 1983)

1970 - The PLATO IV project got under way to demonstrate the technical feasibility, manageability, and economic viability of an extensive computer based education (CBE) network. The system used a large time-shared computer with as many as 600 high resolution plasma screen remote terminals with touch panels.

Courseware was created with a new implementation of the TUTOR language which had grown to 236 commands. (Alessi & Trollip, 1985; Kearsley et al., 1983a; Lockard et al., 1987; Sherwood, 1977)

- The TICCET (Time-shared Interactive Computer-Controlled Educational Television) system was launched by the MITRE Corporation. It supported up to 128 student stations consisting of a keyboard, TV set and headphones from a mini-computer. Eventually renamed TICCIT (Time-shared Interactive Computer-Controlled Information Television), development was supported by the University of Texas and, in 1972, by Brigham Young University. The system emphasized adult instruction with a great deal of learner control via special function keys. This simplified courseware development since the author did not have to program complex sequencing decisions. Lessons were created using the first example of a non-programming authoring system called APT (Authoring Procedure for TICCIT). (Alessi & Trollip, 1985; Barker, 1987; Chambers & Sprecher, 1983; "History of TICCIT," 1978; Stetten, Morton, & Mayer, 1970)

1971 - CONDUIT was set up to distribute courseware on a large scale. (Chambers & Sprecher, 1983; Kearsley et al., 1983b)

- The Chicago City Schools Project began using Suppes' materials and 850 terminals. (Chambers & Sprecher, 1983)
- The National Science Foundation invested \$10 million in two CAI projects, PLATO IV and TICCIT. (Lockard et al., 1987)

1972 - The Minnesota Educational Computer Consortium (MECC) was created to make computers accessible to the public schools. Its influence eventually grew far beyond the State of Minnesota. (Alessi & Trollip, 1985; Chambers & Sprecher, 1983)

1973 - The National Development Program in Computer Assisted Learning (NDPCAL) began its £2 million five year program in the United Kingdom. Eventually it

encompassed 690 staff members on 35 projects in 47 institutions. (Chambers & Sprecher, 1983; Kearsley et al., 1983b)

Early 1970s - The Huntington Project in New York created simulations in high school science. These were written in BASIC and were widely disseminated, which promoted the use of BASIC in CAI. (Kearsley et al., 1983a)

- CAI research projects began at various centers across Canada: the Ontario Institute for Studies in Education (OISE) at the University of Toronto using a PDP8, Concordia University, Queen's University, the University of Calgary, and the National Research Council (NRC) of Canada with a PDP10. (Chambers & Sprecher, 1983)

1974 - As the result of a collaborative effort of CAI researchers from across Canada and the coordination of the NRC, the NATional Author Language (NATAL) was defined. It was a terminal independent, portable, high-level procedure-oriented language similar to PL/1 with powerful computational, logical, and file-handling facilities. First implemented on NRC's DEC 10 computer, it is now available on a range of main-frame, mini-, and microcomputers. (Barker, 1987; Westrom, 1974)

1.3.3 Theoretical Period [1975-82]

Two seminal events stand out in this period. First was the appearance of stand-alone, ready-to-run microcomputers.

The key ingredients were beginning to accumulate -experience with CAI in different settings using somewhat different instructional models and new hardware technology, which in 1977 saw the introduction of successful microcomputers by Radio Shack, Commodore Business Machines, and Apple Computer. Microcomputers made it possible to *economically* use CAI in educational environments -a practice previously unattainable. (Lockard et al., 1987, p. 148)

Anyone, who desired to, could now do research and development in CAI.

Second, from the field of artificial intelligence in the late 1970s, came the first experiments in intelligent computer-assisted instruction (ICAI) (Alessi & Trollip, 1985; Kearsley et al., 1983b). Instruction was generated by combining a subject-matter knowledge base with a sophisticated model of the learner.

Of fundamental importance to the Theoretical period was the appearance at the beginning of this period of a number of refereed journals dedicated to this area of research:

1. The Association for Educational Data Systems began publishing its *AEDS Journal* in 1967 but the first articles on CAI did not appear until the summer of 1973. An annotated bibliography on CAI was published in the spring of 1974. In 1986 this publication was renamed the *Journal of Research on Computing in Education* and AEDS had become the International Association for Computing in Education.
2. Information Synergy, Inc. began publishing its *Technological Horizons in Education* (T.H.E.) *Journal* in 1974.
3. The Association for the Development of Computer-Based Instructional Systems (ADCIS), which grew out of the IBM 1500 users' group, began publishing its *Journal of Computer-Based Instruction* in August 1974. This association had a special interest group on theory and research.
4. Baywood Publishing Co., Inc. began publishing its *Journal of Educational Technology Systems* in 1972 and began carrying an increasing number of articles on computer-based education in 1975.
5. Pergamon Press Ltd. began publishing its *Computers in Education: An International Journal* in 1976.

Immediately after the close of the Theoretical period two more journals made their appearance:

1. Crane, Russak, and Co. began publishing their *Machine-Mediated Learning: An International Journal* in 1983.
2. Baywood Publishing Co., Inc. began publishing its *Journal of Educational Computing Research* in 1985.

Some other major developments of this period:

Mid 1970s - The US Navy developed their Computer-Managed Instruction (CMI) System. The US Air Force developed their Advanced Instruction System.

PLATO and TICCIT became commercially available. (Alessi & Trollip, 1985)

1976 - The Educational Testing Service, under contract to the National Science foundation, did a major evaluation of both PLATO and TICCIT. ("History of TICCIT," 1978; Murphy & Appel, 1977)

1977 - IBM introduced their Interactive Instruction System (IIS) for their System 370 architecture. This included the Coursewriter III authoring language, the Course Structuring Facility (CSF), and the Simulation Exercise Facility (SEF). (Barker, 1987; *Interactive Instruction System*, 1977)

1978 - Control Data released an updated version of the original TUTOR language for their PLATO System. TUTOR had now grown from its original 70 commands to 290. Although, this number of commands could be compared to those of some modern structured BASIC implementations for microcomputers with over 300 commands. (Control Data, 1978; Engels, Görgens, & Ostrowski, 1988)

1979 - Computer adaptive or tailored testing was introduced. A unique test is generated for each student from a stratified test item pool. Each successive item was dependent on the answer to the previous item. Such a system could determine a student's level of achievement performance with a very much shorter test.

Late 1970s - Research into learner control was under way. The major question was how much learner control and what kind. The best answer seemed to be the mixed initiative dialogues typical of ICAI systems. (Kearsley et al., 1983b)

1980 - The United Kingdom's NDPCAL was followed up with the five year £9 million Microelectronics Education Project (MEP) which supported 130 individual projects. (Chambers & Sprecher, 1983; Kearsley et al., 1983b)

1981 - PASS (Professional Authoring Software System), one of the first of the powerful authoring systems for microcomputers, was developed by Bell & Howell for the Apple II. (Barker, 1987; Bell & Howell, 1981)

- The 19 campuses of the California State University System commenced a coordinated CAI program based on the Apple II microcomputer and the PASS authoring system. (Chambers & Sprecher, 1983)

- The IBM Personal Computer was introduced supported by CP/M-86, UCSD p-System, PC/DOS, and MS/DOS. (Lemmons, 1981)

Early 1980s - Research and development in computer-controlled video instruction started. (Alessi & Trollip, 1985)

- Research into the motivating qualities and instructional characteristics of simulations and games was undertaken. (Kearsley et al., 1983b)

Kearsley et al. (1983b), on reviewing over 50 major CBI projects in terms of their theoretical and practical significance, concluded:

CBI has been primarily driven by advances in computer and information systems technology while instructional theory has lagged behind applications.... It should be clear that CBI rests upon a broad foundation of research even though it frequently appears to be almost totally atheoretical and pragmatic in nature. (pp. 95-96)

There were nine major outcomes from their (1983a) study:

1. There is ample evidence that computers can make instruction more efficient or effective.

2. We know relatively little about how to individualize instruction.
3. We do not have a good understanding of the effects of instructional variables such as graphics, speech, motion, or humor.
4. A great deal has been learned about overcoming institutional and organizational inertia and resistance to change in the context of implementing CBI.
5. Significant progress has been made on the development of authoring tools and techniques for CBI.
6. Good mechanisms have been developed for the dissemination of CBI ideas and courseware.
7. CBI has spurred research throughout the entire field of instruction.
8. [United States] federal funding has played a pivotal role in advancing CBI.
9. We have just scratched the surface of what can be accomplished with computers in education.

It is concluded that CBI research has had substantial impact on education when assessed in total. An even greater impact is anticipated in future decades as CBI technology becomes more powerful, accessible and prevalent.
(p. 90)

1.3.4 Automation period [1983-90]

Within this period there has been a tremendous increase in the number of computer users. Computer-based training (CBT) has expanded in industry along with the use of personal computers. Stein and Staiti (1988) gave descriptions of 51 authoring systems for the IBM PC family of microcomputers, two for the Apple II, three for the Macintosh, five for mini-based systems, and four for mainframe-based systems. They also described 12 available authoring languages. Barker (1987) describes over 60 languages and systems used in computer-based education and training.

With more powerful personal computers being produced in this period, ever more powerful authoring tools are being provided the CAI/CBT researcher and user. For

the CAI author or perspective author many CAI how-to texts have appeared during this period (Alessi & Trollip, 1985; Chambers & Sprecher, 1983; Godfrey & Sterling, 1982; Walker & Hess, 1984). More and more colleges, universities, and educational systems are offering credit and non-credit courses in CAI (Collis & Muir, 1984; Glenn & Carrier, 1989). Ever more widening horizons of research are now being pursued (Marchionini, 1988; Niemiec & Walberg, 1989; Park, 1988). It is perhaps too early to decide which events and activities were pivotal to this period.

1.4 Aspects of Computer-Assisted Instruction

CAI is often perceived as having two major components: content and control. (New systems often mention a student model component, but this could be viewed as a special part of the control component.) These components might be further broken down into, what may be termed, aspects of CAI. The aspects in the content component are display, student input, and answer analysis. The aspects in the control component are sequence control, computation, and courseware management.

The display aspect covers the presentation of content to the student. This could include text and graphics on a CRT, video motion and still sequences on a monitor, prerecorded or computer generated voice or music, or computer controlled laboratory instruments. The student input aspect encompasses communicating with the computer through a keyboard, keypad, special function keys, mouse, joystick, specially designed switch panel, or laboratory instruments. The answer analysis aspect can be quite complex. It includes prompting the student to respond, accepting the response, editing and analyzing the response, categorizing the response, and finally taking some action based on the response category.

The sequence control aspect used to mean taking decisions to branch, or GOTO, a labeled statement in the courseware. In a modern structured control environment it would include the use of looping and decision structures, and the referencing of

named service routines, procedures, and modules with the ability to pass parameters. This aspect controls the path each student takes through a course and bases decisions on the value of variables from the computation aspect. Some of these variables could be part of a complex student model. The computation aspect includes all predefined system data types and variables, and all operations on these types. This aspect should also provide for the definition of user defined variables of system types, and the definition of user defined types and variables and the operations on these types. The assignment of values to variables is also part of this aspect. The courseware management aspect provides for the storage and management of all components of a course. In large courseware projects with many and varied elements produced by a team of authors, this may become quite complex. This aspect also defines the scope and visibility of all elements.

There is some cross-over between the content and control components. The courseware management aspect may be related to the general organization of the content, and the answer analysis aspect usually includes a local intrinsic branching structure that controls sequencing of feedback to the student and perhaps some score keeping computations.

NATAL was probably the first CAI language to specifically incorporate, in its structure, the two components of CAI. The designers of most authoring systems also recognize this division, but do not provide the same power and flexibility in the control component as do the better authoring languages. Because of this lack of power and flexibility in the control component many authoring systems are ill-suited for the development of some modes of CAI.

1.5 Future Needs

As new hardware systems, including local and wide area networks, and software operating environments are developed, new opportunities for the creation of advanced

CAI systems will evolve. Barker (1987), after reviewing many of the authoring tools available, has suggested the following requirements for a "General Purpose Authoring Environment (GPAE)" for CAL/CBT:

1. Ability to operate in both an autonomous mode and a networking environment
2. Provision of support for a variety of human-computer interaction techniques
3. Ability to cater for both line and frame oriented dialogues
4. Provision of good frame-editing facilities
5. Cater for the provision of graphics, animation, and audio
6. Capable of providing highly end-user oriented interfaces, that is, it must be user friendly and easy to use
7. Capable of providing adequate database and knowledge base support facilities
8. Able to capture broadcast (radio and TV) material from global distribution systems
9. Provides support for a variety of information storage media
10. Able to create dynamic models of individual students and to use these to produce highly individualized instructional schemes
11. Capable of incorporating standards (where they exist) in order to facilitate the exchange of instructional material
12. Ability to produce generic material that can be easily modified to meet different requirements
13. Capable of providing an extensible environment that is able to accommodate unforeseen advances in technology and user requirements
14. Capable of being made highly reliable and of being easily maintained by local technicians
15. Able to provide facilities to support parallel simultaneous tasks (through multi-tasking and multi-processing)
16. Able to support inter-task communication with parallel activities
17. Capable of providing facile control of the learner's interaction environment.

(p. 229)

Such systems will be needed to more fully support requirements for integrated individualized learning systems in education and demands for re-training systems in industry.

2. The Problem and Expectations

2.1 The Problem

Brahan, Farley, and Orchard (1985) defined three main goals in the development of courseware production tools that have motivated researchers since the inception of CAI:

1. To provide for increased productivity by using the computer to reduce the demands on the time of the course designer,
2. To permit the course designer to make effective use of the computer without having to develop complex skills relating to the details of programming and computer architecture,
3. To improve the transportability and adaptability of the courseware. (p. 1)

They traced this development of courseware production tools through four generations:

- o The first generation is represented by the assembly languages that were used for the initial experimental work.
- o The second generation consists of the specialized CAL authoring languages that provided features directly related to the application, but were still closely tied to the machine with concise notation and restrictions on flexibility.
- o The third generation saw the introduction of the procedure-based [CAL] languages and structured design principles. These languages ... provide for a clear expression of the courseware design through the language itself, and have good communication with the user. At the same time, the user is provided with a high degree of flexibility in accessing to the capabilities of the computer.
- o The fourth generation represents a departure from the traditional approach to programming with a move towards authoring systems as opposed to authoring languages. Through the use of application oriented program generators, the task of programming is transferred from the course-author to the computer. Through the use of predefined strategies and data templates, the course-author can be completely unburdened from the task of computer programming. (p. 2)

Brahan et al. (1985) went on to comment that these authoring systems often restrict flexibility and "do not permit the use of strategies that were not incorporated by the system designer" (p. 2), but they do ease the introduction of the course-author to the computer system. However, "the innovative course-authors will have a

tendency to want to push the system beyond its limits as soon as they become familiar with the potential of the medium" (p. 2). Thus a well designed authoring environment should be capable of accommodating the developing skills and requirements of the user.

2.1.1 Evolution of Sequence Control and Courseware Management

As CAI authoring tools evolved over the four generations, each aspect of CAI (display, student input, answer analysis, computation, sequence control, and courseware management [see section 1.4]) was enhanced. The first generation tools, being machine specific assembly languages, had no special features expressly designed to assist the CAI author. It was not until languages like Coursewriter II (International Business Machines [IBM], 1968) for the IBM 1500 CAI System and TUTOR (Avner and Tenczar, 1969) for the PLATO System appeared that these aspects were specifically addressed.

Since this thesis pertains to the sequence control and courseware management aspects of CAI, the evolution of authoring tools dealing with them will be traced through the second, third, and fourth generations. Representative authoring languages from the second and third generation and authoring systems from the fourth have been selected for in-depth study.

Each language and authoring system will be examined to see what types of data were supported, how data were accessed, what courseware management system was provided, and what kinds of control mechanisms were incorporated. Data are used by all of the aspects of CAI but in particular are used to make decisions in the control aspect and effect the flexibility of that aspect. Some languages and authoring systems provide system variables which are automatically updated by the system. They reflect system status and student related response information. The author may use them to make decisions about course flow strategy or to record student progress.

The intrinsic control mechanisms built into the answer analysis aspect will not be examined.

2.1.2 Second Generation Authoring Languages

These languages were exemplified by Coursewriter II and TUTOR. They were, for the most part, machine specific languages. Coursewriter survives as Coursewriter III as part of IBM's mainframe authoring facilities, Interactive Instruction System (IBM, 1977), and in a modified version written in Elf (Education Language Facility, see Appendix A) on a DEC VAX 11/780 at the University of Alberta (Hunka, 1986, 1988a, 1988b). (The Elf/VAX system was recently decommissioned and a new version of Elf is under development for the Apple Macintosh II.) TUTOR is still an active language on W.R. Roach and Associates' (formerly Control Data's) PLATO system and in a modified form in the PC/MS DOS world as TenCORE by Paul Tenczar, the original designer of TUTOR (Klass, 1984).

Coursewriter II

Writing courseware in Coursewriter II was similar to writing assembler code. Each line started with a two letter mnemonic opcode, with an optional one letter modifier, followed by a number of numeric or text parameters. Yet very substantial multi-hour tutorial, drill and practice, and simulation courses were created using this language.

Data Support

Coursewriter II provided three system data types: 100 character string, 16 bit integer, and boolean. The author could not define user types nor were variables available. Each instructional station was allocated core storage for six 100-position buffers, designated b0 to b5; 31 counters, designated c0 to c30; and 32 switches,

designated s0 to s31. The storage locations b0, c0, and s31 had special system functions and so could be overwritten by the system. A student's response was automatically placed in b0, c0 was used to keep track of the student's response time, and s31 was set to indicate a restart condition. If extra switches were needed, any unused counter could be converted to 16 switches. For example, the switches obtained from counter 4 would be referenced as s4a, s4b, s4c, ... s4p.

Courseware Management

Logically, a Coursewriter *course* was made up of one or more *segments*, each segment comprised one or more *lessons*, and each lesson could contain one or more *problems* (IBM, 1968, p. 83). A segment had the physical restriction that it had to reside on one disk cartridge and thus was limited to 20 000 to 25 000 statements. With later improved disk capacity, this restriction was essentially removed. Each segment was given a number from 0 to 127. This number was only a label for the segment and did not imply an order of presentation. Within a segment the beginning of a lesson was considered a logical restart point and was marked by the *prr* (problem restart) opcode. A student who signed off in the middle of a lesson was always restarted at the previously encountered *prr* statement. Normally, a problem could contain content presentation and one question and always began with a *pr* (problem start) opcode. A problem ended with the next *pr* statement or with an explicit *ea* (end answer) opcode. Lessons and problems did not need to be named but any instruction location could be identified by a *label*, made up of 1 to 6 alphanumeric characters. Each label had to be unique within a segment. The system always provided the course name as the label of the first statement of each segment of a course.

Control Mechanisms

Within a problem a powerful implicit branching mechanism was provided. For controlling a student's progress through a segment, an explicit branching statement could be used. A branch could be conditional or unconditional. The condition could be based on whether a switch was set or reset or on the comparison of the value in a counter with that in another counter or of an integer. The target of a branch could be an actual label or a label that had been loaded into one of six return registers, designated rr0 to rr5. This system allowed for a very primitive type of subroutine support with no parameter passing. A branch could also be made to the beginning of the current problem or to the nth next problem as well as to the last executed *enter and process* (ep) statement. To move a student to another segment of the course an unconditional *transfer* statement was used. This statement took as arguments a target segment number and a target label within that segment.

TUTOR

TUTOR code, in many ways, resembled FORTRAN code. It was line oriented. Each line had an English like command word which had to begin in column 1 and a tag field which began in column 9, and line labels had to begin with a numeric. The tag could consist of zero or more arguments separated by commas. TUTOR started out with about 70 commands in the version used on PLATO III but as its use became more widespread and more demands were made for more features it gradually grew to almost 300 commands (Control Data, 1978). The language became very powerful but very complex. This placed a severe memory load on the CAI author/programmer. In the 1980's a number of authoring systems were written in TUTOR to assist CAI authors in developing courseware on PLATO (Bagnall, Szabo, Halls, & Jensen, 1984).

TUTOR was developed in an environment in which computing resources were scarce. The PLATO system was designed to support a large number of terminals,

therefore "the amount of data representing the state of any single user [was] deliberately kept quite small" (Schwartz, 1983, p. 15).

In the following descriptions the PLATO convention of writing TUTOR command words between two hyphens [-define-] will be used. In TUTOR source code the hyphens were not used.

Data Support

Data storage was closely tied to the machine and required extensive familiarity with machine level concepts for the author/programmer to make full use of the variable facilities of TUTOR. Each user was allocated 150 sixty-bit machine words of storage. If the user was registered on the system as a student then these 150 words were saved between sessions. Each word could contain a signed 59-bit integer, a floating point number (or real) with a signed 48-bit mantissa and a signed 10-bit exponent, or 10 six-bit character codes. An upper-case character took the space of two six-bit codes (shift + letter).

Each word could be referenced by two primitive names, `nxx` if it was to be interpreted as an integer and `vxx` if it was to be interpreted as a real. The `xx` was replaced by a number between 1 and 150 inclusive. These locations could be indexed by the syntax `n(<expression>)` or `v(<expression>)`. A string literal of up to ten characters could be assigned to a word. If it was enclosed in double-quotes ("") it would be right-justified in the word and if it was enclosed in single-quotes (') it would be left-justified. Longer strings could be assigned across word boundaries by use of the `-pack-` instruction. This took as arguments the starting location, the string length, and the string to be stored. Care had to be taken not to overwrite other data.

Meaningful names of up to seven characters could be assigned to a memory location by using the `-define-` instruction. This instruction could also be used to

assign variables for student use. Variable definitions were created in named sets. Up to five sets of variables could be active at one time. Sets could be selectively purged. The -define- instruction could also be used to give names to constant values.

Arrays could also be defined as having zero, one, or two dimensions with specified upper and lower bounds. They were defined over contiguous words of storage and again care had to be exercised to not overwrite other data. Each element in the array was a computer word. Authors could define other types of arrays that used partial words or could have more than two dimensions but the rules for the use of these arrays were quite complex and required a sophisticated knowledge of computer data storage format and structures.

Each lesson (what constitutes a lesson is defined below) had a bank of 1500 words of temporary storage per user. These were referenced as ncxx or vcxx, where xx was replaced by a number from 1 to 1500 inclusive. Common data storage could also be defined with one copy available for all on-line users of a particular lesson. This could be specified as temporary and not saved between sessions, or as permanent which could be saved and restored. There were, however, many complex restrictions.

A router (what constitutes a router is defined below) could establish up to 50 router words of storage per student. The primitive names of these router variables were nrxx or vrxx. Again, the xx was replaced by a number from 1 to 50 inclusive. These required careful setup and management by the author and could only be altered in the router. They could, however, be read by instructional lessons called from the router.

Courseware Management

A file of TUTOR source code was referred to as a lesson. It was the basic object of courseware in the PLATO system. A lesson was automatically compiled

('condensed' in PLATO terminology) just before it was executed if a condensed version with a later date/time stamp than the source code file did not exist.

A lesson had both a physical and a logical structure. Physically a lesson was a sequence of TUTOR instructions grouped into named units. The names had a maximum of eight characters. A unit was made up of either or both of two parts. The first part could contain presentation or calculation instructions, usually referred to as regular instructions. The second part was the response judging part which used an intrinsic branching mechanism similar to Coursewriter II. Statements in this part were referred to as judging instructions.

Logically the units were classified by use as main units and auxiliary units. Main units could be formed into a sequence of units while auxiliary units could not and were called from main units as subroutines.

Main units were further classified as either base units or help units. The base sequence of units was considered the main flow of the lesson and progression through a base sequence was under the control of the author. A lesson could, however, have more than one base sequence. A help sequence was entered from a base unit by the student pressing one of the help function keys. Other help sequences could be called from within a help sequence but when the help sequence was terminated either by reaching its logical end or by the student's request, the student was returned to the base unit from which the original help request was made and that base unit was re-executed.

Some units had special functions. The physically first unit of the lesson if it had no unit header instruction was designated the Initial Entry Unit. It was executed each time a student entered the lesson and could be used by authors for lesson initialization. The -imain- instruction could be used by authors to specify an auxiliary unit that would be executed as an initialization procedure every time a main unit was entered. The -finish- instruction was used to specify a close down unit that

would be executed if a student quit an instructional session by pressing the SHIFT-STOP key.

If a TUTOR course had more than one lesson a router could be used to control the sequence of lessons designated as part of the course. A standardized system router was provided and maintained by system programmers. If this was inadequate for an author's needs then a customized router could be written. This required careful setup and management by the author. A student was automatically sent to the router at sign-on and returned there whenever a lesson was completed.

Control Mechanisms

In TUTOR, branching was considered either author-initiated or student-initiated. In author-initiated branching the decisions were made by code written by the author. Student-initiated branching could be by selection from a list of options provided by the author or by the pressing of a labeled function key the destination of which was set up by the author.

The function keys were classified as those that controlled main unit sequencing (NEXT, BACK, and STOP) or those that selected a help sequence (HELP, LAB, and DATA). Each key could also be used with the SHIFT key thus giving 12 possible function selections. The author could attach a unit destination to each function key. For example, to set up the NEXT key the author would use the -next- instruction followed by the destination unit name. To set up the SHIFT-NEXT key combination the -next1- instruction was used. There were three default settings. The NEXT key would take the student to the next physical unit in the lesson. The SHIFT-BACK key would cause the student to leave a help sequence and return to the base unit from which the help sequence was called. And the SHIFT-STOP key initiated a request to interrupt the current lesson and return to the router or the lesson from which the current lesson was called.

Another type of student-initiated branch was with the TERM key. An author could place in a unit a -term- instruction followed by a term-name. Anywhere in the lesson the student could press the TERM key, type the term-name, and be taken to the unit containing that -term- instruction. This was controlled like a help sequence call. A unit containing a -term- instruction without a term-name was considered the default term unit. A student would be taken to this unit if the term-name typed did not match any term-names in the lesson.

The author could use the -base- instruction to change a help sequence into a new base sequence or to return a student to a different base sequence.

Branching instructions could be classified as within-unit branching, between-unit branching, or between-lesson branching. Most instructions with a specific destination had a conditional and a non-conditional form. For example, the -branch- instruction was one of the within-unit branching instructions that normally took as an argument a line-label to branch to unconditionally. The conditional form of this instruction was:

```
branch expression,label1,label2,label3,label4,...
```

When this instruction was executed the expression was evaluated. If the result was negative then the branch was to label1, if zero to label2, if one to label3, if two to label4, etc. In TUTOR boolean TRUE was equal to -1 and FALSE was equal to 0. Therefore if the expression was boolean then there could only be two labels, label1 for TRUE and label2 for FALSE.

Three within-unit multi-line control structures were supported. Nesting of control structures was permitted. There was an if ... elseif ... else ... endif structure with the -else- being optional and the -elseif- being optional and repeatable. The following counting loop syntax was available:

doto label,index <= from,to,by

<from>, <to>, and <by> where expressions and <by> was optional. It was permitted to branch into and out of a -doto- loop. The line containing the label ended the counting loop structure. The index was the counter variable.

A more generalized loop ... reloop ... outloop ... endloop structure was supported. The -loop- and -endloop- were required. The -loop- instruction could be followed by an optional condition. While the condition was true the loop would be executed again. The -reloop- was followed by a condition which if true would cause a branch back to the -loop- instruction. The -outloop- was followed by a condition which if true would cause the loop to terminate.

There were four between-unit branching instructions. Each could take a conditional form. The -jump- instruction was used to go to a new main unit. The -goto- instruction was the same as the -jump- instruction except the new unit was not initialized. The -join- instruction was used to call an auxiliary unit. These calls could go up to 10 levels deep. The -do- instruction was similar to the -doto- instruction except the target was an auxiliary unit that was to be iterated a counted number of times.

A unit could have up to 10 formal parameters that were all call-by-value and were not local. A call to a unit could have less actual parameters than the called unit had formal parameters.

There were three termination instructions. An -exit- instruction was used to terminate an auxiliary unit structure, an -end- instruction was used to terminate a help sequence, and an -end lesson- instruction was used to terminate a lesson.

The -restart- instruction could be used by the author to specify the lesson and the unit in which the student would restart execution if the current lesson was

terminated before completion. This restart location could be the current unit, another unit in the current lesson, or another lesson and unit. Normally the system would set this to be the first unit of the current lesson.

-jumpout- was the only between lesson branch instruction. It could be conditional and permitted the author to specify a lesson and, optionally, the unit within that lesson where the student was to go. This instruction could also be used for four special destinations. The *resume* destination would send a student to the restart lesson and unit. The *router* destination would return the student to the router. The *return* destination would return the student to the first unit in the lesson from which the current lesson was entered. And the *return,return* destination would return the student to the lesson and unit following the unit from which the current lesson was entered.

In the router the -route- instruction was used to specify which units in the router were to act as reentry units when the student exited from one of the instructional lessons in the router. The router had, as a minimum, four units. The Initial Entry Unit was executed at session sign-on time. The *end-lesson* unit was executed after sign-on or upon return to the router on normal completion of a lesson. The *execution-error* unit was executed if an error occurred in the lesson the student was in. And the *finish* unit was executed when a student requested sign-off from the router. The *end-lesson* unit could allow the student to choose a lesson from the router's index of lessons or it could use a logic table when branching decisions were to be made based on the student's performance and place in the curriculum.

2.1.3 Third Generation Authoring Languages

These languages were exemplified by NATAL, Canada's NATional Authoring Language. This was a procedure oriented, structured language similar to PL/1 that was designed to be implemented on a wide range of hardware from main frames to

personal computers and to be relatively terminal independent. NATAL was a fairly compact language of about 35 statements with a Display Sub-Language of 33 commands.

Originally defined as NATAL-74 under the auspices of the National Research Council of Canada (Westrom, 1974), it was eventually commercialized as NATAL-II (Honeywell, 1981a, 1981b) and implemented on a number of mainframe and minicomputers. Microcomputer implementations were eventually developed as microNATAL (Pressman & Pressman, 1986) and Alpha NATAL (Brahan & Godfrey, 1984). The latter runs under the COHERENT and UNIX operating environments and remains commercially active. The NATAL-II version will be described below and later compared to the Alpha NATAL version.

NATAL-II

A NATAL program (or course) was created in a source code file by a system text editor. NATAL statements could be entered in free form and lines could be indented to show the internal structure of the course. For ease of development and editing the course code could be created as independently compiled modules.

The NATAL course preparation system, PRENATAL, was used to assemble an executable course. Source modules were CHECKED for syntax and semantic errors and if error free were compiled into an Intermediate Language Code (ILC) and placed in a course directory or a library directory. A directory contained a set of NATAL routines (what constitutes a routine is explained below) in ILC and a list of routine names referenced from these routines. A course object file could be PREPARED from a course directory by assembling all the routines in that directory and any others needed from the NATAL System Library and any available User Libraries to resolve all references. The course object file was in ILC which could be interpreted by the NATAL Delivery System.

This process was a tedious way for authors to develop courseware. The module the author was creating or editing could not be viewed in student mode until it was CHECKed (compiled) and PREPARED (linked) in the PRENATAL system. The NATAL Delivery System then had to be entered and the entire course run.

Data Support

NATAL was a loosely typed language similar to *APL*. Variables (name length up to 20 characters) could change both value and type by assignment. The types supported were switch (boolean), switch vector, numeric (signed integer and real), numeric vector, string (maximum length of 2000 characters), file, and label. A variable that had not been assigned a value was considered undefined and if referenced would cause an error.

A vector was a dynamic one-dimensional array that could change length by assignment, again like *APL*. A vector literal took the form (n,n,n,...), where n was a numeric literal or switch literal depending on the vector type. Switch literals were TRUE and FALSE. A pseudo vector could appear in a vector expression. Each element in a pseudo vector was either a literal, an expression, a scalar variable, or a vector variable, for example (3,2*A,N). A and N might reference a scalar or a vector. An element of a vector could be referenced by subscripting of the form variable.(index). Index had to be an integer expression greater than zero and less than or equal to the length of the vector.

A file variable was associated with a particular file in the underlying operating system and then used for all internal references to that file.

Any NATAL statement could be prefixed with a label. Labels could be assigned to label variables which had to be prefixed with an ampersand, for example, &NEXT <- PROB#3. Label variables could be targets in a GOTO statement, for example,

GOTO & NEXT would be equivalent to GOTO PROB#3. Label variables could not change type.

A course author could define up to 60 course variables (prefixed with #) that were accessible by all students executing the same course. These variables could be switch, numeric and string only. There was also a number of system variables provided.

Except for course and system variables which were global, all other variables were local to the routine within which they were defined. To be referenced in other routines they had to be passed as parameters. This could make parameter lists quite long.

Courseware Management

A NATAL course was made up of a set of named routines. Routine definitions could not be nested, thus routine names were global to the course. In source code, a routine began with a labeled header statement followed by a possible formal parameter list. The label was the routine name and was referred to as an external label. A routine ended with an END statement.

Routines were divided into three classes: units, procedures, and functions. A unit provided a complete instructional transaction. It was used to present course material to the student, to request a response, to categorize the response, and then to supply appropriate feedback. Execution of a unit provided a powerful intrinsic branching mechanism that was driven by the response categories. A unit's name was prefixed by an asterisk (*).

Procedures controlled the flow of instruction within a course. Calculations, file I/O, performance recording, and logical decisions were made in procedures. Procedures could call other procedures and units. This provided a hierarchical structure of control for a course. A course was entered through an ENTRY

PROCEDURE. Its label was the course name and it could not have parameters. There could only be one ENTRY PROCEDURE in a course.

Functions played a supporting role. They could be invoked by units, procedures, and other functions to support their actions. Functions were classed as general, edit, or graphic functions. General functions were invoked in expressions and returned an explicit result which could be of type switch, switch vector, numeric, numeric vector, or string. An edit function was used to edit a student's response and was invoked by its name appearing in the argument list of a unit's EDIT statement. A graphic function performed drawing actions on the display screen and could be invoked by its name appearing in the argument list of an &G display command or a PLOT statement.

NATAL did not impose a predefined structure to a course. However, if an author or instructor wanted to use the system provided performance recording, analysis, and reporting facilities, the course had to be divided into logical lessons that were given number labels of up to three digits in length. Within lessons, each question had to be given a number label of up to five digits in length.

Control Mechanisms

The following control mechanisms could be used in procedures and functions but not in units.

Statement labels inside routines were local to the routine and were termed internal labels. They could be the target of a GOTO statement in the same routine. Labels could be indexed. This took the form label#int, where int had to be a literal integer greater than zero, for example, PROB#5. An indexed label could be referenced as GOTO label#(exp), where exp was an integer expression. For example, GOTO PROB#(N+2) would be interpreted as GOTO PROB#5, if N was equal to 3.

The CALL statement was used to invoke a procedure or a unit. It was not allowed in functions or units. The target of a CALL could not be a label variable.

Both procedures and units could have indexed labels, for example, LESSON#2. Thus a CALL LESSON#(CURRENT-4) would be the same as CALL LESSON#2 if CURRENT was equal to 6. There was no facility to call other NATAL courses or system programs from within a course.

All routines except ENTRY PROCEDURES and EDIT FUNCTIONS could have formal parameters which had to be unsubscripted variables and could not be course or system variables. The actual parameters in the invocation of a routine could be either an unsubscripted variable, which was then treated as a call-by-reference, or an expression, which was treated as a call-by-value. If there were fewer actual parameters than formal parameters, the extra formal parameters were undefined. If there were more actual parameters than formal parameters, the extra actual parameters were ignored.

The CHECKPOINT statement was used to set a restart position for a student. On passing a CHECKPOINT, the system recorded in the student's restart record the course location and all the system variables and conditions. If the student later signed-off, he or she could be restarted at the statement following the CHECKPOINT. An EXIT statement did exactly the same thing except it forced the student to sign-off.

The DO-Block was a powerful control structure in NATAL which took four forms. All DO-Blocks started with a DO statement and ended with an END statement. In its simplest form with just a DO header it was used to collect a set of statements so they could be treated as a single statement. The second form took the syntax:

DO variable = spec1, spec2, ...

Each specx could take the forms: exp1 [TO exp2 [BY exp3]]. Each expx was a numeric expression. If just exp1 was specified then the variable was assigned the

value of `exp1` while the block was executed once. The other forms took the normal counted iteration semantic for the block. After `spec1` was executed then `spec2` was carried out and so on until the list was exhausted. The third form took the syntax:

```
DO WHILE (logical-expression)
```

While the logical expression was `TRUE` the block would keep iterating. The fourth form took the syntax:

```
DO variable = spec1, spec2, ... WHILE (logical-expression)
```

This was executed exactly like the second form except the logical expression was checked at the beginning of every iteration and if `FALSE` execution of the block would terminate.

The decision structure took the following syntax:

```
IF logical-expression THEN group1 [ELSE group2]
```

`Group1` and `group2` could either be a single statement or a `DO-Block`. These control structures could be nested.

Condition Blocks could be specified in procedures, general functions, and graphic functions. They specified actions to be taken when the indicated condition was raised. The syntax was:

```
ON CONDITION condition action
```


Action could be a single statement or a DO-Block. Some of the conditions that could be watched for were: ATTN, response category, ENDCOURSE, ENDFILE, ENTRY to procedures or units, ERROR, INPUT('string'), RESPONSE, and RESTART. Some conditions could be restricted to a specified routine. Conditions could be DISABLED and later ENABLED. A Condition Block would be active as long as the routine within which it was specified remained active.

Alpha NATAL

Brahan and Godfrey (1984) state:

Alpha NATAL is a completely distinct implementation of the original NATAL Language Specification, designed to work within COHERENT or UNIX operating systems. The Softwords implementation is written in C and functions on IBM XT, DEC's PDP 11 series, and Motorola 68000 hardware.

Alpha NATAL includes a number of improvements and amendments to the NATAL specification suggested by various users and by a design group at work on details of that specification. ...

Most of Alpha NATAL source code (on average, 95%), will look just like NATAL II code. (app. E-1)

Some features that make Alpha NATAL distinct from NATAL II are summarized below.

Alpha NATAL is compiled to relocatable object code instead of ILC which required a NATAL interpreter. It is also strongly typed, thus all variables must be declared by type before use. The INTEGER and INTEGER VECTOR types are also supported. The LABEL type is not supported. Variables may be declared as global in scope. Indexed labels for statements and routines are not supported but similar actions can be defined using the provided CASE structure.

Libraries of routines in source code can be accessed by using an INCLUDE statement in the course source code file. All formal parameters are call-by-value, call-by-reference is not provided. Edit and graphic functions are treated like general

functions and have no syntactic difference. Also Alpha NATAL permits functions to call procedures and units. The general Condition Block structure of NATAL II is not supported. ON ATTN can be used to service interrupts.

The NATAL II parallel registration and performance recording, analysis, and reporting facilities are not provided for in Alpha NATAL. Regular file I/O and operating system facilities would be used to replace these. The CHECKPOINT and EXIT statements are also not supported.

2.1.4 Fourth Generation Authoring Systems

Stein and Staiti (1988) described 80 current CBT (computer-based training) authoring systems and languages. Most of the authoring systems were designed to operate on the ubiquitous IBM PC family of microcomputers, although there were two reported for the Apple II and three for the Macintosh micros. Minicomputers and mainframes each had five authoring systems listed. Twenty-eight of the authoring systems were menu driven, 30 were menu driven with access to an underlying authoring language, two were icon driven, and four were icon driven with access to an underlying authoring language.

Stein and Staiti also listed six types of programming tools provided by some authoring systems:

1. unresolved branch checker - automatically checks all branches; reports any that lead to a dead end
2. font editor
3. sequence editor for flowcharting
4. alternate character sets - scientific and/or foreign languages
5. random test generator - selects test questions from a bank at random
6. student tracking and reporting facility (1988, p. 6).

Only 12 of these authoring systems were reported to possess all six of the above programming tools. Two of these were chosen for this analysis. PCD3 (Control Data, 1987) is a menu driven system and operates on the IBM PC family of microcomputers. *Authorware Professional* (Authorware, 1987, 1989) is icon driven and operates on the Macintosh family of microcomputers.

PCD3

Originally developed by Control Data as an authoring system for the PLATO CAI system, PCD3 (PLATO Courseware Design, Development, and Delivery) was eventually only marketed for the IBM PC family. It is now supported by W.R. Roach and Associates who recently took over CDC's interests in the PLATO system. The 2.0 version is examined here.

PCD3 can be used for the development of individual lessons, entire courses, or a complete curriculum. However, to use the power and resources of all the facilities provided, all the data for the lesson, course, or curriculum must be stored in one PCD3 file, the size of which is limited by the underlying operating system. PCD3 is designed for the easy update and modification of instructional strategy and content materials. Courseware displays and designs are represented visually.

Data Support

PCD3 is a strongly typed language but does not support user defined types or pointers. The number of PCD3 user variables that an author may create is only limited by the available memory. Variables have six characteristics: name, type, initial value, current value, storage class, and description. All variables are global.

The name of a variable can be up to 20 characters long and an author may attach a 40 character description of the variable's use. The storage class may be temporary or permanent. Temporary variables are assigned their initial value when a

student enters a PCD3 file but its value is lost when the student leaves the file.

Permanent variables are assigned their initial value the first time a student enters a PCD3 file but on subsequent entries the variable's previous current value is restored.

The following types are supported: integer (range -32768 to +32767), decimal (range $\pm 10 \times 10^{\pm 306}$), string (maximum of 120 characters), logical (literals TRUE and FALSE), array (only one dimensional), and record. Arrays may be of type integer, decimal, string, logical, and record. The cell index may range from -32768 to +32767. When an array is created the author must specify the number of cells, the starting number, the type, and the one initial value for all cells. Records may have any number of fields. For each field the author must specify a name (up to 20 characters), a type (may be integer, decimal, string, or logical), and an initial value.

Variables of record type may only be created with the variable editor and once created, fields may not be added or deleted. Variables of all other types may be created with the variable editor or, if undefined, when first used in an expression. The variable editor may also be used to edit a variable's name, initial value, current value, and description. However, once created a variable cannot change its type. Variables may be deleted but the system does not check for dangling references. A reference to a deleted variable will cause a run-time error. This can only be corrected by deleting the entire expression containing the illegal reference and then recreating the expression with legal references.

Twenty-eight system variables are provided. Their names are prefixed with the dollar symbol (\$). Only their current values may be edited by the author.

Key name constants may be assigned to integer and decimal variables. Function keys have PLATO type names such as HELP, NEXT, BACK, STOP, DATA, and LAB. Their shifted values are given as SHIFT_NEXT, etc. The IBM PC key pad names such as Home, End, PgUp, and PgDn are also supported. The regular keys are specified in

single quotes as 'a', 'b', 'A', 'B', etc. (String literals are enclosed in double quotes ("...").)

Thirteen standard arithmetic functions are provided as well as six type conversion and character set functions.

Courseware Management

Although the manual professes that PCD3 may be used to create lessons, courses, or curricula, it does not impose any pedagogical nomenclature on the author. The author sees courseware as being divided into content, strategy, and events. The courseware is stored in a PCD3 file which contains both a content database, arranged as a topical outline, and a hierarchically called set of strategy maps.

The system provides three interconnected editors. The author can decide 'What to teach' with the Content Editor, 'When to teach it' with the Strategy Editor, and 'How to teach it' with the Event Editor.

The Content Editor is used to edit the Content Base. Each level of the Content Base may contain *topics*, which are organizational headings that point to an outline at the next lower level, and *components*, which are instructional events. The first level of this database is referred to as a content outline. All subsequent levels are referred to as section outlines. The leaves of this tree structure are *components*. The content component is the smallest element that an author can use separately in a lesson. It can consist of examples, definitions, text, illustrations, questions, commonly used graphics, or programming subroutines.

The Strategy Editor is used to edit the hierarchical base of strategy maps. One of the elements of a strategy map is an *event* which is edited by the Event Editor. An *event* is an instructional episode. Content components are also *events* that are edited by the Event Editor and may contain the same objects. Although an *event* may

reference a *component* it may not modify it. If, however, a *component* is modified the change will be reflected in every *event* that refers to it.

The PCD3 manual states:

There are a number of advantages to keeping the content base separate from the instructional strategy. The PCD3 Authoring System allows an author to isolate content from strategy to achieve several goals:

- o Permit easier reuse of either a content base or a strategy.
- o Provide a tool to organize content regardless of strategy.
- o Provide a library for commonly used text, graphics, or utility routines. (Control Data, 1987, p. 3-50)

Control Mechanisms

The Strategy Editor is used to edit the nodes on a flow chart that visually represents the course flow in a particular strategy. Each strategy has a *main* flow path that goes from the top to the bottom of the screen. There are six node types that may be placed on the *main* flow path:

1. Event Presents instructional material and asks questions. Edited by the Event Editor.
2. Decision Indicates branches based on specified conditions. Initiates a *branch* path from the *main* path of a flow chart.
3. Menu Presents a menu, or index, to the student. Allows the student to select the next strategy. This node also initiates a *branch* path.
4. List Collects nodes attached to the *branch* path of a *menu* or a *decision*. If more than six nodes are needed on a *branch* path they must be grouped into a *list*. For a *menu* a *list* may contain up to 26 nodes, for a *decision* up to 99 nodes. *Event*, *file*, and *strategy* nodes are the only node types that may be placed on a *branch* path or in a *list*.

5. File Allows a link to other PCD3 files, or to files of other applications, by means of a DOS command. Parameter passing is not supported. After exit from a linked file the student returns to the next node in the current strategy map.

6. Strategy Nodes may be grouped and placed into a strategy node. Strategy nodes on the *main* path may be expanded to their underlying nodes. There is, however, a maximum number of nodes that can be on a *main* path. Strategy nodes on a *branch* path may not be expanded.

There are some common options that an author may make at both a menu and a decision node. The author may select to have the node and its associated *branch* path form a loop that repeats N times, repeats until a specified condition becomes TRUE, or repeats until all nodes have been selected at least once. The author can also specify for each node in the *branch* path an availability condition for entry into that node. If this condition is evaluated to FALSE the node cannot be entered if selected.

The selection of one of the nodes on a menu *branch* path is made by the student via a menu *event* associated with the menu node. This *event* may be system generated or built by the author using the Event Editor.

The selection of one of the nodes on a decision *branch* path is controlled by the author through the selection option at the decision node. The author may choose to present each node in sequential order, to present nodes in random order with or without replacement, to select a node by the value of a variable or expression, or to select the first node that has its availability condition set to TRUE.

Under certain options there are some further choices that must be made. If the required repetition of the decision node is not complete and all nodes have been selected, should the student go to the beginning of the list or exit the decision. If the decision node is re-encountered after normal exit from the node should selection be the next available node in the list or should selection begin with the full list.

The author may also set a condition under which a decision node is bypassed or exited before completion. For menu nodes the author can allow the student to select to exit the menu with the SHIFT-NEXT key either at any time or under a set condition.

Nodes are processed from top to bottom on the *main* path and from left to right on the *branch* paths according to settings in the *menu* and *decision* nodes. Menu and event nodes may be set as restart points for students who leave the file before normal completion. They may also have backflow conditions set which allow students to back through the courseware to see previously presented material.

There are seven categories of objects that may be included in an *event*:

1. Graphics Text, graphics, screen and object erases.
2. Logic Labels and end labels for grouping objects, assignment of expressions to variables, if ... otherwise ... endif structures, repeat ... until structures, and comments.
3. Content Reference to a *component* in the Content Base.
4. Query Accept and judge student input and provide feedback. Has an intrinsic branching structure.
5. Pause Wait for key press, screen touch, or elapsed time.
6. Object Code Load and call C routines.
7. Videodisc Control of videodisc player.

An *event* is created using an interactive visual process. When an *event* is listed it

appears as a structured programming language with automatic indentation of *if*, *repeat*, and *query* structures.

Authorware Professional

In December 1987 Authorware, Inc. released their much anticipated authoring system, *Course of Action* version 1.0. They also released an advanced version, *The Best Course of Action*, that supported colour graphics, *movies*, digitized sound, and videodisc control. The chief designer was Dr. Michael Allan who was also responsible for the initial design of PCD3. In September 1989 the version 1.5.1 was renamed. *Course of Action* became *Authorware Academic*, and *The Best Course of Action* became *Authorware Professional*. The aspects reviewed here are common to both versions so the authoring system will just be referred to as *Authorware*.

For authoring, a Macintosh Plus, SE, or II is required with two 800K floppy disk drives as a minimum requirement. However, the use of a hard drive is recommended. The above equipment, or a 512K Macintosh with a minimum of one 800K disk drive, is required for delivery of the courseware to the student. A product called *Authorware PC* is available to convert courseware for delivery by the IBM PC family of microcomputers.

Courseware is designed and developed in a highly visual and interactive environment. The author works in two windows: the presentation window which shows what the student will see, and the design window which shows an annotated flow chart of the section of the courseware being edited. Presentation software is embedded within the authoring system allowing the author to view and edit the courseware at the same time. This ability to edit the courseware directly while it is being presented encourages authors to explore with their design and gradually build from a prototype to a completed product.

We've been careful not to reduce the flexibility of the system in order to make a feature simple. However, ease-of-use is one of our major goals.

The features of *Authorware* are the result of successive efforts and refinements to merge ease-of-use and power. (Authorware, 1989, p. 1-3)

Data Support

User defined variables are global within an *Authorware* file and have the following characteristics: name (up to 40 characters), type, times used (in expressions, etc.), initial value, current value, and description (up to five lines of 40 characters). Only three types are supported. The numerical type (both integer and real) has a stated range of $\pm 1.1 \times 10^{104932}$ but does not support E-notation and has width limitations on displaying values. (The manual states that the exponent is 104932 but Authorware could not confirm if this value was correct.) The character type supports strings of up to 30 000 characters. The logical type supports the boolean literals TRUE / FALSE, 1 / 0, ON / OFF, and YES / NO.

Although variables may not change their type they may be freely used in mixed type expressions and automatic conversion of their values to the target type for the expression will take place. These conversions follow explicit predefined rules.

Variables are created in the *New Variables* window which pops up if either a new variable name is used in an expression, the author selects the "New ..." command of the Variables menu, or the author chooses the New option of the "Show variables ..." command of the Variables menu. Under this last command are five other options: Duplicate, Rename, Delete, Change initial value, and Change current value. If a variable is renamed its new name will automatically appear in each place it is used in the file. A variable may only be deleted if its *Times used* characteristic is zero.

System variables are divided into seven classes (the number of variables in each class is given in parentheses): Question (33), Decision (6), Time (21), General (44), Video (3), Graphics (2) and File (8). The Decision variables will be referred to in the **Control Mechanisms** section below. The author may assign values to 14 specific system variables. Forty-one system variables have values that are local to certain

types of icons (courseware design objects that may be given a local *title*). They retain their local value during execution and this local value may be referenced by the following syntax: `VariableName@"title of icon"`. Without the @ suffix the most recent value is returned.

System functions are divided into eight classes (the number of functions in each class is given in parentheses): Math (19), Character (24), Time (8), Jump (7), Video (4), Graphics (7), General (22), and File (7). Fifty-three of these *functions* do not return a value and, therefore, may not be used in expressions. More properly they should be called procedures. The Jump functions will be referred to in the **Control Mechanisms** section below.

User defined functions (and procedures) may be imported into an *Authorware* file. They must conform to the Apple *HyperCard* format of XCMD for external commands (procedures) and XFCN for external functions. These routines may be loaded from *HyperCard* stacks or *Authorware* user-function documents.

The contents of a variable of type character may be thought of as a set of words delimited by space characters or as a set of lines delimited by <RETURN> characters. The following system functions provide ready access to and manipulation of these data objects: `DeleteLine(string,line#)`, `GetLine(string,line#)`, `GetWord(word#,string)`, `InsertLine(string,line#,newline)`, `LineCount(string)`, `ReplaceLine(string,line#,newline)`, `ReplaceWord(word,replacer,string)`, and `WordCount(string)`.

There is a unique system-provided array available to the author. It is a sparsely populated array that uses indexes that range from 0 to 60 000. Both numbers and character strings may be mixed in the array. Two system functions are provided to give access to the elements of this array: `ArraySet(Index,Value)` and `ArrayGet(Index)`.

Courseware Management

The smallest independent unit of courseware is an *Authorware* file. This is the object that is created and edited by the authoring system, and executed by the presentation software. It contains courseware design icons and their contents, variable definitions, user defined functions, and reusable courseware elements called models. With the exception of the system variable *StudentName*, all user and system variables are local to an *Authorware* file.

It is often necessary to divide the content for a course among multiple *Authorware* files, whether for instructional design reasons or because of memory or disk space limitations. *Authorware* provides the facilities to branch from one file to another and, optionally, to return after completion, and to share specified user variables between files. Large bodies of course content may require elaborate course structures such as the use of a router file to control access to component files or to share common resources among multiple files.

The presentation software automatically updates a student restart record file containing the values of all variables, the graphics of the current display, and the current position in the courseware. One of these record files is kept for each *Authorware* file the student has entered. They are grouped together in a records directory, one directory for each student. The information in these files is used to restart a student at a subsequent session.

Authorware files may be executed from floppy or hard disk drives, network servers, or CD-ROM. The courseware must be "packaged" for student use. For a small course, no larger than about 500K, the presentation software, the *Authorware* file(s), and a student's records directory may be placed on one floppy disk. For a course with no *Authorware* file larger than about 750K, the presentation software and a student's records directory could be placed on a student disk and all the *Authorware* files placed on any number of other floppy disks. These files could be

copied to a hard drive. Another option is to have all files on a hard drive, network server, or CD-ROM. Of course, the students' records directories could not be on the CD-ROM. This arrangement would allow for *Authorware* files to be larger than 750K.

Predefined courseware structures of design icons called models may be loaded into an *Authorware* file for use by the author. An author may create and save models to be loaded into any *Authorware* file. These models may be pasted anywhere in a file from a pulldown menu. They may then be modified as needed. Any variables that are used within a model are automatically created, if they do not already exist, when the model is pasted in the file. If a variable of the same name already exists the author may choose to have the variable from the model renamed by appending a 2 to the variable's name.

Control Mechanisms

Control in an *Authorware* system may be divided into three groups: within-file control, between-file control, and student-start/restart control.

Within-file control is managed in a design window. Running from the top to the bottom of a design window is a flow line upon which the author may paste design icons. Each icon denotes a special function which is performed when it is encountered during interaction with a student. An icon can be given a title which may have up to 400 characters. Listed below are the eight basic design icons provided by the *Authorware* system:

- o **Display icons** put text and/or graphics on the screen.
- o **Animation icons** move the object(s) of a preceding Display icon from one point to another in a given amount of time or at a specified speed.
- o **Erase icons** erase the text and/or graphics displays.
- o **Wait icons** interrupt file flow until: 1) the user presses a key or clicks the mouse, or 2) a specified amount of time elapses.

- o **Decision icons** select which icon(s) from a set of attached icons to use next.
- o **Interaction icons** present options or questions and then, based on the user's response, select and branch to attached icons for feedback to the user.
- o **Calculation icons** perform arithmetic or special control functions, execute user-written code, jump to other files, or jump to other applications.
- o **Map icons** organize and modularize the file by providing space to put more icons. Each Map icon provides its own flow line on which you can place other icons, including Map icons. (Authorware, 1989, p. 3-5)

Authorware Professional provides additional icons which include Sound, Movie, and Video icons.

Decision and Interaction icons have attached icons pasted on a microbranching flow line. Each attached icon represents a separate branch path. An unlimited number of these branch paths may be attached. If there are more than five the *titles* are scrolled in a window thus showing a maximum of five icons and their titles at one time. The branch paths visually show both the selection and optional looping structures selected by the author.

Decision and Interaction icons must be placed in a Map icon before they can be attached to another Decision or Interaction icon.

The Interaction icon, its microbranching flow line, and its attached icons form the intrinsic branching logic associated with the answer aspect of CAI.

When the author opens a Decision icon, options for repetition, selection, and time limit are presented. If the author wants the structure to repeat there are four options: repeat a set number of times (specified as a number, a numeric variable, or an expression), repeat until all branch paths have been selected at least once, repeat until a key is pressed or the mouse button is clicked, or repeat until a specified conditional expression becomes TRUE.

There are four selection options. The *sequential* option specifies that each path will be taken in order from left to right. An internal counter keeps track of this

count while the student is in the file. The *random without replacement* option randomly selects a path, but all paths must be selected at least once before any may be repeated. The *random with replacement* option selects a path with each path having an equal chance of selection each time. Lastly, the *pick nth path* option evaluates an expression the value of which determines the path selected. If the value is less than one or greater than the number of paths, the Decision icon is bypassed regardless of the repeat option. For conditional expressions TRUE = 1 and FALSE = 0.

The time limit option allows the author to specify an expression the value of which specifies the time limit in seconds for the Decision icon to complete all tasks regardless of the depth of decent through attached Map icons. There is an option to display an alarm clock that graphically shows the amount of time left. The author may also attach a Display icon with the title "TimeLimit" that will automatically display a message if timeout occurs.

There are six Decision system variables that are automatically updated and which the author may make use of in expressions. They are all of the multi-value type so may take the @"title" suffix. These variables are listed below:

RepCount	The current number of repetitions by a Decision icon.
SelectedEver	For each path, set TRUE if the path was previously selected.
AllSelected	Set TRUE if all paths have been taken at least once.
PathSelected	The number of the last path selected.
TimesSelected	For each path, the number of times the path has been taken.
PathCount	The number of paths not counting the "TimeLimit" icon.

There are two system functions that can be used for within-file control.

Functions are called within a Calculation icon. The Restart() function causes an immediate branch to the beginning of the file and initializes all variables. The GoTo() function requires the system variable IconID. IconID@"title" returns a unique numeric identifier for the icon whose "title" is specified. Thus GoTo(IconID@"title") will cause an immediate branch to the specified icon.

The flow line in the design window has limited capacity so icons must be grouped and placed in Map icons. Map icons on a flow line can also be ungrouped if there is room for the contained icons. Map icons on a microbranch path may not be ungrouped unless they contain only one icon. The decision to limit the number of icons on a flow line was taken by the designers to force modularization. There is no limit to the nesting of Map icons.

Between-file control is by system functions that are called from within a Calculation icon. The JumpFile() function is used to transfer control to another *Authorware* file. The JumpFileReturn() function acts the same way except on exit from the called file, control is returned to the next icon in the calling file. The syntax for these functions is:

```
JumpFile[Return](["file"["varspec"["RecordsDirectory"]]])
```

The parameters must be string literals or character variables. The first parameter is the name of the target *Authorware* file. If this parameter is not present the student must choose from a standard "Open file" dialogue. The second parameter is a list of user variables to be passed by name to the called file. (Files do not have formal parameters.) The asterisk may be used as a wild card character so that a set of variables with the same prefix may be passed, for example, shared*. System variables may not be passed but their values may be placed in user variables to be passed. The

last parameter allows the name of the student's records directory to be specified. It is only by a JumpFile call that a student's records directory may be changed.

The JumpOut() and JumpOutReturn() functions are used to pass control to another application. The syntax for these functions is:

```
JumpOut[Return](["application"],["document"])
```

The parameters must be string literals or character variables. Either the first or the second or both parameters must be present. The first parameter is the name of the application to be called. The second parameter, if present, is the name of the document that is to be opened by the application. If the first parameter is not present then the application that created the document named in the second parameter is called.

Student-start/restart control is managed by a few system variables and functions. The author may use the system variable Resume to control the placement of the restart points in a file. When a file is reentered by a student it is automatically resumed at the last restart point recorded in the student's records directory.

The system function Quit([mode]) can be used by the author to force the student to exit a file. If mode is not present or is 0 then the student is returned to the file that called the current file. Mode 1 returns the student to the Finder (the operating system), mode 2 restarts the computer, and mode 3 turns the computer's power off. The system function QuitRestart([mode]) acts the same as the function Quit() except the restart record is set to start the file from the beginning and to reinitialize all variables.

The function ResumeFile() can be used to restart the student at the icon following the Calculation icon containing a Quit(mode) function or at the beginning of a file that was exited by the QuitRestart(mode) function. In both cases the mode had

to be greater than zero. The form, `ResumeFile("recordfolder")`, will resume in the file specified by the named restart record. The function `ResumeFileName(["recordfolder"])` will return the name of the file to be resumed.

To handle many students on the same computer or on a network the author may elect to use a Startup file to control student signon to the courseware. Once the student is identified the author can use a `JumpFileReturn()` function to send the student to a course router file using the third parameter to specify that student's records directory. On return to the Startup file after exiting the router, the student's normal sign-off procedure could be handled. Authorware provides example Startup and Router files that may be used as models for building custom designed files.

The presentation software may be started automatically. It will look first for an *Authorware* file called "Startup". If this is not found it will look for one called "Router". If neither is found, and on all floppy drives it can find only one *Authorware* file, it will open and run that file. If it finds many *Authorware* files, these will be presented to the student in a list for one to be selected.

2.1.5 Beyond Lessonware

After examining these authoring languages and systems it becomes apparent that the focus of the languages is on the individual instructional transaction and the control and management of sets of these transactions. The size of these sets is usually limited by practical considerations such as available computer memory or file size limits. This may tend to make courseware fit the pedagogical form of a lesson, that which a student would usually complete in one sitting.

The design of *Lessonware* is usually where the beginning author gains first experience. But from the earliest developments of CAI, the CAI authoring languages and systems have always been pushed to produce larger courseware products. This

has usually meant the addition of facilities such as router lessons that controlled the invocation of the elements of the larger course. The languages and systems tended to provide a two level system of management and control: a within-file system and a separate and distinct between-file system.

A typical example was observed by the writer at the Division of Educational Research Services, Faculty of Education, University of Alberta, in the summers of 1989 and 1990. A CAI project was undertaken to develop tutorial, practice, and testing modules for a complete grade 12 mathematics program. This followed the Alberta Mathematics 30 curriculum which divided the year's work into six units which, in turn, were divided into a total of 40 topics.

The project made use of nine CAI authors for varying lengths of time. The CAI programming experience of these authors ranged from nil to 20 years. The system chosen for the project was *Authorware Professional*. This system proved excellent for the design and development of individual lessons.

Generally, each topic had an *interactive CAI resource* (tutorial) and an *interactive test bank resource* (practice) both created with *Authorware*. A *written test bank resource* was provided to the teacher via a regular test banking system called *LRX Test*. Each of the CAI modules (tutorial and practice) were created as separate *Authorware* files. Individual topics were assigned to each author. Almost 40 megabytes of courseware were developed.

Since *Authorware* does not directly provide student navigation features for between-file or within-file movement, these had to be developed by the authoring team. This required the use of shared resources and extensive human coordination between individual projects.

Most difficulties arose because of the need to manage resources, both human and computer. This included the sharing and updating of common resources, ensuring the proper management and backup of various versions of each of the *Authorware* files,

and the coordination of file updates. Although *Authorware Professional* was an excellent tool for the development of lessons, it had few features for assisting with meta-level coordination.

In large scale courseware projects there is a need to provide for the organization and management of the courseware and its supporting data within a complete development environment. This should include the ability to use locally defined data objects.

2.1.6 Hierarchical Control and Management

Because of the limitations of human memory, mankind has always needed to organize and classify knowledge. This often leads to a hierarchical organization. One only needs to look at the organization of the curricula at a university or the table of contents of a textbook to see examples of this need. The Mathematics 30 CAI project is typical of the need to organize and classify material. The course was divided into units, units into topics, topics into major headings, and major headings into *pages* (sets of instructional transactions or practice questions treated as basic elements).

Better organization and management of large scale courseware projects would be facilitated if a hierarchical courseware database and management system with a unified control system were provided. The control system would allow for many levels of control but with the same structure and commands, thus completely doing away with the two-level (within-file, between-file) non-unified systems in present use. This hierarchical structure could also form the basis of scope and visibility rules of the named entities of the system, thus reducing the risk of naming conflicts and the need for enforced naming conventions. All components of this system would have to reside on and be controlled by one computer system. This might be a centrally controlled mainframe or minicomputer with terminals, a distributed system of microcomputers on a local or wide area network, or some combination of both.

Since a hierarchical structure may be applied to many situations, the system should be generalizable to organizing and managing lessons, courses, or curricula, as well as the sub-parts of a lesson.

Although the proposed demonstration project of this thesis research is for a single author environment, the hierarchical structure could be mapped into a user registration system that could control access to the structure. Different classes of users, such as managers, authors, instructors, and students, would have different privileges. For a large courseware project involving many authors, this would facilitate management of the project by assigning various sub-trees of the structure to different authors while still providing them with shared course components and facilities.

2.1.7 Authoring Environments

The highly visual and interactive authoring environment of the *Authorware* CAI system proved popular and easy to learn and use by authors in the development of individual lessons of the Mathematics 30 project.

Large scale courseware projects can become quite complex. This can lead to memory overload for the authors developing the courseware. The authors need to keep track of such things as user defined data elements, instructional strategies used, content already covered and yet to be covered, and the contents of libraries of graphics, audio, and animations. To facilitate the design, creation, debugging, modification and management of courseware, and to reduce the author's memory load, a visual authoring environment is proposed.

Authorware's visual environment is based on two windows, the presentation window which shows exactly what the student will see, and the design window which provides the author with an annotated flow chart of control over lesson flow. This environment worked well at the lesson level but was a less useful tool for organizing

and managing courseware at higher levels of abstraction. Guidelines for the design of a visual authoring environment at these higher levels of abstraction might be sought from the field of human-computer interface design. "Many of these guidelines," Brown (1988) states, "have developed from expert judgment, common sense, and practical experience." However, he continues:

Experimental results on which to base user interface design recommendations are relatively scarce. Existing experimental data are often too situation-specific to provide general design guidance. (p. 2)

The interface design should be such as to assist the user to gain an accurate and useful mental model of the system's organization and operation (Brown, 1988).

2.2 Expectations

Given a predefined CAI authoring language, the task is to define and implement the following: (1) a hierarchical tree structured database to hold the courseware modules and data specified for the given language, and (2) a visual authoring environment to support the creation and modification of the objects contained in this database. For this research project the authoring environment will not include the creation and modification of the language statements or the data items other than what is needed to demonstrate the viability of the system.

While the system design developed in chapter 4 will be for a multi-user environment, the demonstration prototypes described in chapter 5 will be limited to a single user environment.

The authoring language to be used is the one defined and implemented by Chiu, Garraway, Higham, McGinnis, and Nesbit as part of the Educational Research Authoring System (ERAS) project under Hunka (1988a). The ERAS language is particularly well suited for this research as it is designed to have its language modules and supporting data definitions reside in a hierarchical database. Since the sub-language interpreters decode statements that are coded in highly efficient internal

data structures (rather than in source code), the creation and modification of these data structures is easily amenable to a visual authoring environment.

2.2.1 The ERAS Authoring Language

The ERAS language is made up of six sub-languages, each with its own interpreter. The interpreters execute language statements which are stored in internal data structures collected in named modules. The sub-languages are termed: Control, Content, Display, Input, Answer, and Menu. For example, a Control Module contains Control Language statements which are executed by the Control Language interpreter, a Content Module contains Content Language statements which are executed by the Content Language interpreter, and so on. There are also two types of service modules each written in the Control Language: Routine Modules, which may be called to perform a subroutine or procedure, and Function Modules, which may be called from expressions and return an explicit value.

The ERAS language also supports named user defined data type definitions, variables of some specified type, and named constants. Modules may have formal parameters of both call-by-value and call-by-reference. In addition, the Display Language may reference special data storage modules for text, drawings, pictures, and graphs.

A Control Module may call Control Modules at the next lower scope level, and visible Content Modules. Routine Modules may call visible Content Modules. A Content or Answer Module may call visible Content, Display, Input, Menu, and Answer Modules and may have unnamed internal blocks of Display, Input, Menu, and Answer statements. All modules may call visible Routine Modules. Expressions may call visible Function Modules.

2.2.2 The Hierarchical Courseware Database

It is assumed that a predefined system provides for the registration of a course author and the assignment to this author of a course database containing a predefined course entry node and a root node for the course. The course entry node, termed an ENTRY_LEVEL, should also provide the author with course level documentation facilities.

The database should be a multi-branch tree structure. The limit to the number of branches from any node or the total size of the tree is implementation dependent. At any node, the author should be able to create any number of branches from that node. During authoring, branches should be traversable in either direction. A branch should be deletable only if the node that subtends it has no branches. That is, pruning the tree will always be from leaf nodes upwards. A branch should be movable from one node to another at any level in the tree.

In this research project the external generic name for a tree database is "Course". The predefined registration system provides the Course with a given name. Internally, the author should be able to give each level of the tree a generic name and each node in the tree a given name. While the generic name for level 0 (the root level) is predefined as "Course", the author could define the generic names for other levels, for example, level 1, "Chapter"; level 2, "Topic"; and level 3, "Lesson". Thus a Course would contain Chapters, Chapters would contain Topics, and Topics would contain Lessons; and each Chapter, Topic and Lesson would have its own descriptive given name. Generic names should be able to have aliases. For example, "Test" could be an alias for "Topic". Then a Chapter could contain Topics and Tests. The author should be able to create, modify, and delete these generic names at any time. The generic name definitions should be accessed at the ENTRY_LEVEL. The default generic name for level n should be "Level_n".

The set of branches from each node should have a logical order, that is, first, second, third, etc. This order should be determined by the author. New branches should be insertable at any place in the order and the order of existing branches should be changeable at any time. The current order may be referenced by cardinal number. Thus, if the generic name for level 1 is "Chapter", the set of branches from level 0 may be referred to as Chapter 1, Chapter 2, Chapter 3, and so on. A Chapter may also be referenced by its given name. The cardinal number of the ordering should be unique for each generic name and its aliases. For example, the ordered branches of a Chapter might be Topic 1, Topic 2, Test 1, Topic 3, Topic 4, and Test 2.

Each node in the tree is a data object termed a LEVEL. As can be seen in Figure 1, each LEVEL data object points to a unique Control Module for that node and, if there are branches from the node, a "Next Level" directory that points to these branches. A LEVEL may also point to a unique set of directories associated with that LEVEL. The named objects pointed to by these directories are unique to that LEVEL. Table 2 lists the names of the directories that may be in the set and the type of objects each points to. A particular directory should only exist if there is at least one object pointed to by the directory. A directory should be automatically created when the first object for that directory is created and automatically deleted when the last object in that directory is deleted. The names of objects in a directory listing should have a logical order which is decided by the author. That is, an author should be able to create a new object and insert its name anywhere in the order and should be able to change the order at any time. An object should be movable from one directory to any other directory of the same type anywhere in the tree.

The scope and visibility of any identifier should be determined by the location of the object it names in the tree. For example, a variable defined at the root (Course)

Figure 1. LEVEL Directories

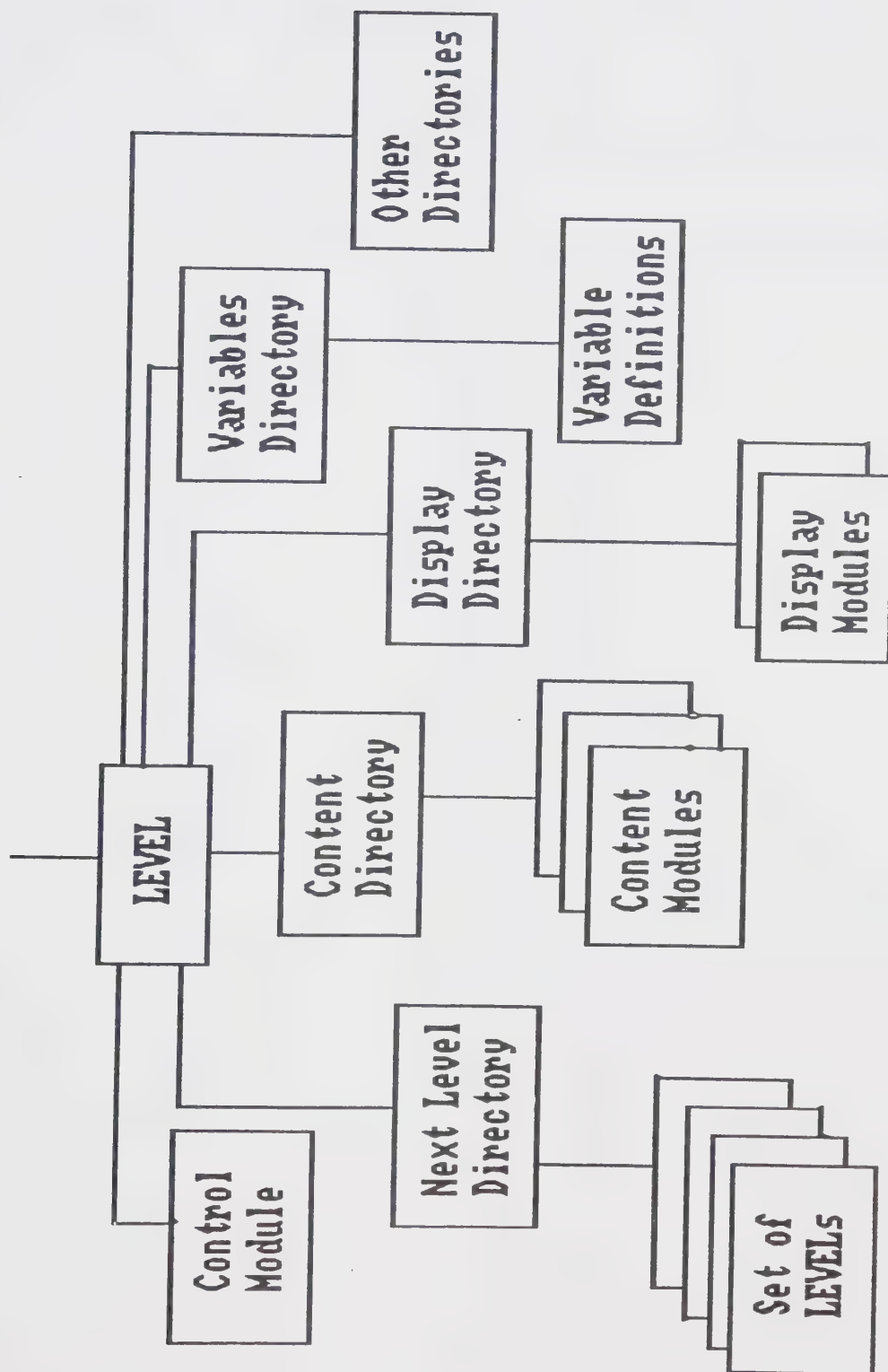


Table 2. Types of Directories Supported

DIRECTORY	OBJECT
Content	Content Modules
Display	Display Modules
Input	Input Modules
Answer	Answer Modules
Menu	Menu Modules
Routines	Routine Modules
Functions	Function Modules
Types	Data Type Definitions
Variables	Variable Definitions
Constants	Constant Definitions
Text	Text Data Modules
Drawings	Drawing Data Modules
Pictures	Picture Data Modules
Graphs	Graph Data Modules

node would have scope over the entire tree. Its visibility could be limited by the presence of another variable of the same name being defined lower in the tree. The variable at the root node should not be visible at the node where the variable with the same name was defined or at any node descended from that node. That is, when an identifier is referenced it should be searched for at the current node and, if not found, should be looked for at the next higher node in the tree. This should continue until the identifier is located or the root node search fails. (Predefined system identifiers are considered to exist above the root node. Therefore, after a failed search at the root node, a search of system identifiers would be made.) There should be no conflict if identifiers naming different object types have the same name. For example, a variable and a content module could have the same name. However objects referenced in expressions (i.e., variables, constants, and functions) must have unique names.

Internal documentation text should be optionally attachable to each Level. This text could be created, modified, or destroyed by the author at any time.

2.2.3 The Visual Authoring Environment

Two basic visual forms are needed to represent the course database for interactive authoring. One is a movable window over a tree diagram that represents the organization of the course. The author should be able to move the window to view any section of the tree. An icon representing each node of interest should be displayed. The icon should include the generic name, logical order number, and the given name for the node. Any visible node should be selectable for modification, deletion (if it is a leaf node), or movement to another point in the tree. When a node is moved, the sub-tree for which it is the root node should move with it. A new node should be insertable before a selected node. However, new nodes may not be inserted at level 0 since there can only be one root node. A daughter node could

be created for a selected leaf node. When a node is created, the author should be interrogated for its given name. If that tree level has any aliases for its generic name, the author must then select the desired alias or default to the base generic name.

The second visual form is a file card metaphor where each card represents a Level data object displayed in a cascade of file cards illustrating the relative position of the Level data objects in the database. When a node is selected for modification, a file card representing the node's Level data object should be added to the cascade of file cards on the screen. This card should display the following data:

- generic name
- logical order number
- given name
- version number
- creation date/time stamp
- last modified date/time stamp

The card should also display selection buttons for:

- the control module
- next level directory (branches from this node)
- content directory
- display directory
- input directory
- answer directory
- menu directory
- types directory
- variables directory
- constants directory
- functions directory

- routines directory
- text directory
- drawings directory
- pictures directory
- graphs directory
- documentation text

When a directory is selected a directory file card should appear. The author could select to have the names appear in logical or alphabetical order. An item should be selectable for modification, deletion, logical order change, or movement to another directory of the same type in another node. A new item should be insertable before the current item. The author should then be interrogated for its given name.

Authoring functions could be accomplished by either or both of two methods. The method(s) available are implementation specific. One method uses a mouse controlled screen pointer, selection buttons, and pull down menus. The other method uses cursor movement controlled by arrow cursor control keys and a function selection keypad. Menus should display only currently available operations. A diagram of the keypad should be displayed on the screen showing the name of the function each active key will perform.

Depending on the operation being performed, the author should be able to move directly to the current directory file card, the current level file card, any visible file card, the root level file card, the course tree display, the entry level file card, or the system.

During an authoring session a run-time environment should be maintained. At any time during editing the author should be able to execute the course from the current location. Conversely, during the execution of the course, the author should be able to interrupt execution, move up and down the execution stack, and select to

edit the course at any level of the execution stack. For debugging purposes, the contents of variables should be examinable at any stack level.

3. Contributing Ideas

Many disciplines and fields of research have contributed to and influenced the design of the CAI system proposed in this thesis. Certainly the examination of the historical development of CAI languages and systems, and experience with them, played a large role. The discipline of Computer Science has also played a part. This chapter presents a short summary of ideas that have proved helpful to the writer. These are the concept of abstraction, visual programming, human-computer interaction, and graphical user interfaces.

3.1 The Concept of Abstraction

The importance of the concept of abstraction in modern software engineering and in the more recent languages such as Modula-2 and Ada has been discussed widely (Berzins, Grey, & Naumann, 1986; Ford & Wiener, 1985; Gannon, Hamlet, & Mills, 1987; Kamel, 1987; Parnas, Clements, & Weiss, 1985; Verity, 1987).

A major approach to problem solving is abstraction, which involves model building. Because people manage complexity by means of abstraction, abstractions provide a view of the essential components and processes that define a system.

Typically, abstractions deal with objects and operations on the objects. High level abstractions are not concerned with implementation details. For example, one may understand the concept of an automobile brake without caring whether it is a drum brake or disk brake. The abstraction (concept) of "braking" is independent of its implementation. Our ability to separate the high level abstractions that we use to view the system from the implementation details (lower level abstractions) allows us to understand complex systems. (Ford and Wiener, 1985, p. 12)

A CAI author who is about to develop a course in, say, mathematics may decide to break up the course into topics (higher level abstractions) without yet being concerned about the details of content presentation (lower level abstractions). Later, as work is being done on a particular topic, the author may have to consider the

order of presentation of content (higher level abstractions) before being concerned with actual screen displays (lower level abstractions).

The process of abstraction may be described as identifying essential concepts while ignoring inessential details. It is one of the most powerful intellectual tools for dealing with complexity. (Ford & Wiener, 1985, p. 168)

Ford and Wiener (1985) described the concept of abstraction in the context of using modern programming languages to solve problems. Problem abstraction means problem decomposition, the "process of reducing a large problem to a set of smaller more manageable problems" (p. 4). The evolution of problem abstraction is tied closely to the history of programming languages and the software development methodologies associated with these languages. It has been an evolution away from the machine.

When using machine language the programmer had to think at the machine level. This led to a "bottom-up" approach, where one is first concerned with implementation details. Assembly languages use symbolic names for machine instructions and memory address abstractions, which places the programmer a slight step away from the machine.

FORTRAN and ALGOL represented a "giant leap away from the machine" (Ford and Wiener, 1985, p. 13). They provide much better support for problem abstraction. This includes name abstractions (variables), expression abstraction (operations that combined literals and variables), control abstractions (iteration and conditional branching), data types (static and structured), and functional abstraction (subroutines). These languages allow the programmer to link a problem with high level data and control structures, and lead to the strategy of using flowcharts. However, using the flowchart methodology, there is too much implementation detail too soon.

"Perhaps the most significant contribution of the ALGOL-like languages of the 1960s was the power and subtlety associated with scoping and visibility" (Ford and Wiener, 1985, p. 14). The scope of an identifier may be defined as that "section of a program in which the identifier maintains a particular associated meaning" (p. 111). In general, the scope of an identifier is the program *block* where the identifier is declared unless the same identifier is declared in a nested *block* where it is then excluded. "An entity is said to be *visible* within the scope of its identifier" (p. 112). If an entity is visible it may be used in its usual way. This allows procedures to be written without concern for the context in which they will be used. A system can then be partitioned into components at the subprogram level which could perform well-defined and simple operations.

Pascal, introduced in the early 1970s, "offered the software developer additional power in the form of richer data structures and control abstractions" (Ford and Wiener, 1985, p. 14). Modula-2 and Ada, which appeared in the early 1980s, presented the module (or package), data hiding, strong type checking, concurrent processing, and object-oriented design. These languages could be used at the design level of problem solving. They separated the problem abstraction (conceptional definition) from the implementation.

Software design requires the definition of data objects and the actions to be performed on these objects. Two design methodologies have evolved. They differ in their emphasis, in the early design stage, on the two software components. The first of these methodologies is usually referred to as "top-down" design or structured programming with an emphasis on the actions to be performed and their order. A problem is solved by a series of refinement steps approaching more and more detail until the programming level is reached. The second methodology is referred to as object-oriented design, or data abstraction, with an emphasis on the data objects. In this method the types of data that are important to the solution of the problem are

first determined. The new data types (and the set of operations on each type) are then designed. Lastly, the overall system that makes use of these types is designed. Data abstraction is "one of the newest and most powerful abstractions" (Ford and Wiener, 1985, p. 168). Procedures and functions (functional abstractions) are the building blocks of both methodologies.

To understand data abstraction it is first necessary to understand the concept of data type. A data type may be defined as a set of values and a set of operations on those values. A language may be designated as strongly typed if it allows only operations defined for a particular type to be performed on values of that type. This characteristic often helps programmers catch subtle programming errors. It is important that the representation of a type (implementation details) be hidden. This assures consistency in the use of the type and lowers maintenance costs if the representation is later changed. An example is the array type provided by most programming languages. The programmer does not have access to the implementation of arrays. This is hidden in the compiler. The compiler designer is free to make upgrades to the implementation as long as the interface to the array type is not changed.

For example, Modula-2 uses the library definition and implementation modules to create abstract data types. The definition module for an abstract data type exports the name of an *opaque* type and the interface to all the operations on that type. The implementation of the type definition and of the functions and procedures for the operations are hidden in the parallel implementation module. The opaque type is implemented as a pointer to a base type which may be defined in terms of the predefined types of Modula-2, which may be considered to be low level data abstractions, and the structured type constructors (arrays, records, sets, and pointers) used to build higher level data abstractions.

This approach allows the programmer to design closer to the problem domain and farther from the computer, to define the essential concepts separately from the implementation (data or information hiding). The advantages of this methodology are: 1) the programmer does not have to remember inessential details, 2) it guarantees the integrity of the values of the type, 3) the implementation may be upgraded without the need to recompile the modules that use the type, and 4) related software components are put together to make testing and maintenance easier.

An example of the use of data abstraction in a CAI environment might be the following. The topic of complex numbers is to be taught. The abstract data type for complex numbers could be defined as an *opaque* type and a set of functions and procedures on this type. The CAI author could then develop the tutorial, practice, and testing lessons and use the abstract data type for complex numbers to assist in presenting the concepts and in the answer analysis of questions in the practice and testing lessons. Currently no CAI authoring language or system directly provides this type of facility.

3.2 Visual Programming

The challenge of this decade is to bring computer capabilities, simply and usefully, to people without special training in programming. *Visual Programming* represents a conceptually revolutionary approach to meet this challenge.

Visual programming has gained momentum in recent years primarily because the falling cost of graphics-related hardware and software has made it feasible to use pictures as a means of communicating with computers and to use computer graphics as a medium to teach programming. However, even though work on visual-oriented computing is now mushrooming, there is no consensus on what visual programming is, let alone on a way to assess it. (Shu, 1988, p. v)

During the latter half of the 1980s, the literature regarding approaches to visual programming have been diverse and prolific (Beretta, Mussio, & Protti, 1986; Chang,

1987; Glinert, 1986; Grafton & Ichikawa, 1985; Korfhage, 1986; Matsumura & Tayama, 1986; Raeder, 1985; Tanimoto & Glinert, 1986).

Shu (1988) offered the following definition of visual programming: "the use of meaningful graphic representations in the process of programming" (p. 9). She suggested that one of the motivations behind the development of visual programming is to exploit the nonverbal capabilities of programmers by making better use of the right brain's parallel processing, and intuitive and artistic sense. She sees visual programming stimulated by the following premises:

1. Pictures are more powerful than words as a means of communication. They can convey more meaning in a more concise unit of expression.
2. Pictures aid understanding and remembering.
3. Pictures may provide an incentive for learning to program.
4. Pictures do not have language barriers. When properly designed, they are understood by people regardless of what language they speak.
(p. 9)

After examining the recent work reported in the literature, Shu (1988) offers the following classification scheme for visual programming systems and research:

A. Visual Environment

1. Visualization of
 - a. Data or information about data
 - b. Program and/or execution
 - c. Software design

2. Visual coaching

B. Visual Languages

1. For handling visual information
2. For supporting visual interactions
3. For actually programming with visual expressions (Visual Programming Languages)

- a. Diagrammatic systems
- b. Iconic systems
- c. Form systems

Some systems fit into more than one of these categories.

Ideas from all of these categories could apply to some aspect of a full CAI system. Specific research activities reported by Shu (1988) that influenced the design and implementation of the CAI system developed for this thesis are given below.

From the category *Visualization of Data* (A.1.a.) of Shu's classification scheme, the Spatial Data Management System built at the Computer Corporation of America is a technique to access data from a large data base through graphical representations. This system utilizes a visual browsing facility designed for non-programmers. The user can approach the data base from a "world view", a high-level global view, and selectively obtain ever more detailed views of specific data.

INCENSE, a system for displaying data structures, was developed by Brad Myers at the Xerox Palo Alto Research Centre. It is used to display Pascal-like data structures in a manner similar to what a programmer might draw on paper. The user can make custom layout designs to display data structures or accept the system default layouts. Dynamic data structures use pointers graphically to effectively show the data relationships.

Visualization of Programs (A.1.b.) is needed to help the programmer understand programs and is required for the complete programming life cycle of designing, coding, debugging, testing, and modifying. Pretty-printing, which is the use of indentation, well placed comments, and white space, has long been a standard way for visually bringing out the program structure and making programs more readable. The *SEE visual compiler* by Baecker and Marcus (Shu, 1988) employs graphical design principles and laser printer technology to produce neatly enhanced layouts of C programs.

PECAN, a family of program development systems which supports multiple windows, is being developed at Brown University. Based on Pascal, it uses structured templates for building programs, immediate feedback of semantic and syntactic errors while editing, and multiple concurrent views of a program. Internally, programs are stored in an abstract syntax tree. The user never sees the tree directly, but sees views, or concrete representations, called "semantic views". These include the symbol table, data type definitions, expression trees, control flow graphs, and a syntax-directed editor. For program execution there is an execution control view which emits messages that indicate the current execution state of the program plus program input and output. For debugging, each statement being executed can be highlighted via the syntax-directed editor and the flow graph view. Also the execution stack can be displayed showing the current stack frame, the variables in that frame, and their values.

Shu (1988) introduces her chapter on *Visualization of Software Design* (A.1.c.) with the following statement:

An understanding of specifications, design decisions, system structures, dependencies among data and components, etc., are crucial throughout the software life cycle, and become increasingly difficult to grasp as the software increases in size and complexity. This observation, together with the success of using graphical techniques for the visualization of "programming in the small," has stimulated the efforts at visualization of "programming in the large." The primary goal is to provide a visual environment that supports the whole software life cycle for large, complex software systems. (p. 67)

The Computer Corporation of America has undertaken a project to design and implement a Program Visualization system. The project has defined ten categories of illustrations that can be of use throughout the software life cycle:

1. System requirements diagrams
2. Program function diagrams
3. Program structure diagrams
4. Communication protocol diagrams

5. Composed and typeset program text
6. Program comments and commentaries
7. Diagrams of flow of control
8. Diagrams of structured data
9. Diagrams of persistent data
10. Diagrams of the program in the host environment. (Shu, 1988, p. 68)

Of most interest to this thesis project is the use of graphical displays to illustrate system architecture. Relational diagrams show the entire data base of the system. By using a windowing system, the user may zoom-in on a box of interest and display the next level of detail. This may continue to as many levels as is needed to extract or build the system data.

Shu (1988) defined a *visual programming language* (B.3.) as "a language which uses some visual representations (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional language" (p. 138). She goes on to qualify this definition. "*The language itself* must employ some meaningful (i.e., not merely decorative) visual expressions as a means of programming" (p. 139). Shu found that "based on the principles of design, most of the visual programming languages reported in the literature fall into three broad categories" (p. 140): flow chart and diagram based systems (B.3.a.), icon based systems (B.3.b.), and form based systems (B.3.c.).

Traditional programming goes through an iterative process of problem analysis, charting (diagrammatic depiction), coding, translating (compiling or interpreting), and testing. The diagrams from the charting phase help the programmer in the coding and testing phases. They are also useful in program documentation. However, there is one major problem. As coding changes are made, as a result of testing, the diagrams soon get out of step with the code. The solution; make the diagrams executable. This should make programs easier to comprehend, document and maintain.

Shu (1988) reported on a number of such systems. Of interest to this research are FPL and Pascal/HSD. *FPL (First Programming Language)* was developed by Robert Taylor at Columbia University's Teachers College as an instructional language. A group of symbols were designed for a subset of Pascal statements. A student uses these symbols to interactively build a flowchart for a program. The symbols have to be supplemented with textual information. The FPL flowchart can then be translated into Pascal for execution.

Pascal/HSD (*hierarchical structured diagrams*) was reported by Diaz-Herrera and Flude and cited by Shu (1988). This system uses two logic symbols, an oval shape for an *ACTION* and a rectangle with pointed ends for a *TEST*. These symbols can be put together in various combinations on a vertical flow path to represent all Pascal structures. An *ACTION* symbol can also represent an abstraction for another flow path of symbols. *ACTION* symbols then nest all actions until a leaf *ACTION* symbol is encountered which represents a Pascal statement.

The Xerox Star System is the grandfather of most current *Graphical User Interfaces (GUIs)*, at least those that are termed desktop user interfaces (see section 3.4 below). It is an example of an iconic system (B.3.b.). This system introduced the concept of using a metaphor, an analogy with something familiar, say office objects, as the basis for human-computer interaction. After much research and discussion, Xerox decided on the now familiar desktop metaphor with documents, folders, file drawers, printers, in-baskets, out-baskets, etc., depicted as small pictures, or icons.

Interaction is with a mouse and command keys. A Star object is selected by pointing with a mouse and clicking one of the mouse buttons. The selected icon is highlighted. An action can then be initiated on the selected icon by pressing one of the command keys. For example, if the user presses the OPEN key to open a document, a window will appear where its contents are displayed.

There are two classes of icons, data icons and function icons. Data icons represent objects on which actions are performed: documents, folders, and record files. Function icons perform actions. For example, a file drawer icon gives access to a network file server, in/out-baskets are used for electronic mail. Another feature is the option sheet which displays options for a command and allows these to be changed.

As Shu (1988) pointed out, iconic systems are not the answer to all human-computer interface problems. For example, FORMAL, a form based system (B.3.c.) "does not appear to be as exciting as the iconic systems. But FORMAL is practical, powerful, and concise" (p. 288). On the other hand, iconic languages appear to be good for teaching programming concepts to novices. They are easier to learn and more interesting than conventional, text-based languages. However, the user still must understand the underlying programming concepts.

Iconic languages can become messy and unwieldy as the size of the applications gets large. Compared to linear notation, iconic systems take up a lot of screen space. Researchers are just beginning to address this problem. However, the key problem is to invent suitable and meaningful visual representations for programming constructs. The current popular systems, like the Macintosh, focus on the use of icons to replace commands and menus at the command language level.

3.3 Human-Computer Interaction

During the 1980s there was considerable interest and research published in the area human-computer interaction (Draper & Norman, 1985; Gaines & Shaw, 1986a; Galitz, 1985; Shneiderman, 1987; Thimbleby, 1984). However, for this thesis research C.M. Brown's (1988) *Human-Computer Interface Design Guidelines* were most useful. He stated that his guidelines were drawn from diverse sources, including

1. evidence from experiments,
2. predictions from theories of human performance,
3. principles of cognitive psychology,
4. principles of ergonomic design, and
5. evidence gathered through engineering experience (p. 2).

Good human-computer interface design, C.M. Brown (1988) suggests, should assist users to reduce the amount of mental processing required to use the computer to complete a task. Working towards this goal, some functions are best handled by the computer while others by the human user. Brown listed five *rules of thumb* for the allocation of functions:

1. Reduce the amount of memorization of commands, codes, syntax, and rules required of the user. For example, permit users to select from a list of displayed options rather than entering memorized command strings.
2. Reduce the amount of mental manipulation of data required of the user. Present data, messages, and prompts in clear and directly usable form.
3. Reduce the requirements for the user to enter data. If information is available to the system, or if the design can make this available to the system, do not require the user to enter it manually. Structure dialogues so that manual user entries are minimized. Selecting from displayed lists instead of entering choices manually is also an effective technique to reduce input requirements.
4. Provide computer aids (such as online checklists, summary displays, and online problem diagnosis) to reduce the amount of mental processing required of the user to remember and execute complex procedures with many steps.
5. Use computer algorithms to pre-process complex, multi-source data and present a composite, integrated view of complex patterns or relationships among many variables. (p. 7)

The user's mental model of system operations must be considered. As a person works with a system, a mental model is built as to how the system works. If the user's model accurately reflects the true nature of the system, then the user will feel

at ease with its use. On the other hand, if the user has an inaccurate model, the user's actions with the system are likely to be error prone. A goal of good interface design is to provide the user with an interface to the system that will assist the user in building an accurate mental model of that system. To reach this goal, a consistent set of conventions needs to be decided upon and adopted.

The designer should make use of commonly held expectations to minimize the learning effort of the user. For example, the use of red colour for alarm signals, yellow for caution, and green for safe takes advantage of the user's association of colour with traffic signals. At the same time, these associations should not contradict the user's expectations, as this will only add confusion to the process.

A critical step in defining the design philosophy for the user interface is to establish the appropriate balance of ease of learning, ease of use, and functionality. Ease of learning is the extent to which a novice user can become proficient in using a system with minimal training and practice. Ease of use is the extent to which the system allows a knowledgeable user to perform tasks with minimal effort (for example, less time, fewer key strokes, or fewer transactions). Functionality is the number and kind of different functions the system can perform. (C.M. Brown, 1988, p. 13)

Brown found that the research in this area supported the conclusion that "the systems that were best for novices (easiest to learn) were also the best systems for experts (easiest to use)" (p. 13). He suggested that four techniques can be used to optimize ease of learning, ease of use, and functionality:

1. Design for novices, experts, and intermittent users. Menus should be available to all users but should not be required. Experts should be able to bypass menus and prompting when desired and use shortcuts. However, the designer cannot assume that any user is an expert for all available functions.
2. Avoid excess functionality. At the design stage, prioritize the candidate functions for the estimated frequency and criticality of their use. Those

functions of lowest priority may have to be dropped or relegated to a secondary path if they would lead to clutter and confusion.

3. Provide multiple paths. This may be accomplished by providing menu bypass command keys, type-ahead keyboard buffers for sequences of commands, user-defined macros of command sequences, and multiple device input for the same command (mouse, function key, or keyboard).
4. Design for progressive disclosure and graceful evolution. The multiple path design should "encourage and support the gradual evolution of a user from a novice to an expert. The fundamentals that a user has to know to perform meaningful, relevant tasks using the system should be learned easily with minimal training and experience" (p. 16). As the user gains confidence, more and more details of the system would be encountered.

Some design features that encourage graceful evolution are:

- a. make fundamental functions easy to learn,
- b. make frequently used functions easy to perform,
- c. encourage experimentation,
- d. minimize the consequences of error through reversible actions, and
- e. use defaults to minimize the number of user selections required to produce the most common or most likely outcome. (p. 16)

C.M. Brown (1988) grouped his design guidelines into chapters and sections. Each guideline is numbered with a chapter number and an item number for ease of reference (2.1 for item 1 in chapter 2). The general outline of Brown's guideline chapters and sections is given below. Chapter 1 is a general introduction and not part of the paradigm, so is not listed.

2. Designing Display Formats
 - a. Reserved display areas (2.1-2.2)
 - b. Consistent conventions (2.3-2.8)
 - c. Alphabetic data (2.9-2.13)

- d. Numeric data (2.14-2.16)
 - e. Alphanumeric codes (2.17-2.20)
 - f. Layout of data (2.21-2.35)
 - g. Lists (2.36-2.42)
 - h. Highlighting (2.43-2.50)
3. Effective Wording
 - a. Abbreviations (3.1-3.13)
 - b. Coding (3.14-3.15)
 - c. Terminology (3.16-3.20)
 - d. Grammar and sentence structure (3.21-3.27)
 4. Colour
 - a. Appropriate uses of colour (4.1-4.7)
 - b. Assigning colours for coding (4.8-4.17)
 - c. Recommended colour code (4.18-4.27)
 5. Graphics
 - a. Effective use of graphics (5.1-5.10)
 - b. Icons (5.11-5.16)
 6. Dialogue Design
 - a. Status information (6.1-6.13)
 - b. Menus (6.14-6.20)
 - c. Soft machine controls (6.21-6.27)
 - d. Commands (6.28-6.42)
 - e. System response time (6.43-6.46)
 7. Data Entry
 - a. Prompts for entries (7.1-7.10)
 - b. Input data format (7.11-7.16)
 - c. Keyboard entries (7.17-7.28)
 8. Control and Display Devices
 - a. Touch sensitive devices (8.1-8.7)
 - b. Mouse (8.8-8.12)
 - c. Fixed function keys (8.13-8.15)
 - d. Program function keys (8.16-8.19)
 - e. Light pen (8.20-8.23)
 - f. Voice entry (8.24-8.28)
 - g. Joystick (8.29)
 - h. Trackball (8.30-8.31)
 - i. Keyboard (8.32-8.36)
 - j. Summary (8.37-8.38)
 9. Error Messages and Online Assistance
 - a. Error correction (9.1-9.9)
 - b. Error messages (9.10-9.25)
 - c. Online guidance (9.26-9.37) (p. vii)

3.4 Graphical User Interfaces

Graphical User Interfaces (GUIs) have been in use for some years. The Xerox 1100 Lisp machine, and the Lisp machines developed at MIT are early examples (Covington, 1991). As was mentioned in section 3.2, the Xerox Star system is considered the first of the desktop GUIs (Shu, 1988). The Star system directly influenced the design of the Apple Lisa and Macintosh computers. Today, almost all computers can support one of the GUI systems that have evolved in the latter half of the 1980s. According to Hayes and Baran (1989), the following have come to be associated with most desktop type GUIs:

- o a pointing device, typically a mouse
- o on-screen menus that can appear or disappear under pointing-device control
- o windows that graphically display what the computer is doing
- o icons that represent files, directories, and so on
- o dialogue boxes, buttons, sliders, check boxes, and a plethora of other graphical widgets that let you tell the computer what to do and how to do it. (p. 250)

Hayes & Baran (1989) indicated that most of the desktop GUIs available today may be classified according to the underlying CPU family and operating system. The following list of GUIs is grouped in this way:

- A. Intel 8088/80286/80386 CPU family
 - 1. MS-DOS
 - a. NewWave
 - b. Windows
 - 2. OS/2 - Presentation Manager
 - 3. Unix - X Window
 - a. CXI
 - b. Motif

- c. DEC-windows
 - d. Open Desktop
 - e. Open Look
- B. Motorola 68000/68020/68030 CPU family
- 1. Unix - NeXT - NextStep
 - 2. Mac OS - Macintosh
 - 3. *AMIGA* OS - Intuition
 - 4. TOS - Atari ST - GEM

Most GUIs share similar features, though they may use different nomenclature. Since *AMIGA*'s Intuition is used to implement the user interface for this thesis research, its features will be described here (Engels, Görgens, & Ostrowski, 1988; Highway, 1989). Version 1.3 of *AMIGA*'s Intuition is described because the newer, more advanced version 2.0 was not available in time for this research.

Intuition supports a two button mouse that controls a three colour pointer on the visual display. The image of the pointer may be changed under program control. The left mouse button is used for selection of objects pointed at and the right mouse button is used with the pull-down menus. All mouse actions have keyboard equivalents.

There are two features of Intuition that enhance its usefulness. Since the underlying operating system, *AMIGA* OS, is single-user multi-tasking, this fact is reflected in Intuition which can support many individual tasks operating in separate windows.

The other feature is that Intuition supports multiple *screens* on the same CRT. Each screen normally fills the entire display (but may be less than full height), and each may have a different resolution and support a different colour palette. Screens may be pushed to the background or brought forward. They may also be pulled down

via the mouse to reveal the screen immediately behind. Screens may be 320 or 640 pixels wide with a vertical resolution of 200 or 400 pixels. The 640 wide screen may have up to 16 of 4096 colours while the 320 wide screen may have up to 64 of 4096 colours. There is a special 'hold-and-modify' (HAM) mode which permits all 4096 colours on a screen at once. Some picture digitizing products now support dithered displays of 100,000 apparent colours. (Dithering is a combination of juxtaposed different coloured dots that creates an illusion of still another single colour.) A screen may have a title and the title's foreground and background colours may be set.

Text and graphics are not written directly to a screen but to *windows* defined for a screen. Windows may vary in size and location, and may overlap. A window may in fact be a scrollable window over a large bit-map. Windows may have a title (which may change when the window is active), and the title's foreground and background colours may be set.

A window may be defined with different attributes. Special *gadgets* (small defined areas of a window selectable with the mouse) may be provided for changing the window size, closing the window, pushing the window behind all other windows, pulling the window to the top of all other windows, and indicating the horizontal and/or vertical size of the window with respect to its bit-map. A window may be made draggable by its title bar, defined without a border, or made a backdrop window which cannot be placed in front of other windows.

A window may have associated with it a pull-down *menu*. The menu titles replace the screen title while the right mouse button is held down. Pointing to one of the menu titles pulls down a list of menu items. An item can be selected by moving the pointer over it and releasing the right mouse button. Any item may have a pop-up menu of sub-items associated with it. The sub-menu pops up when its menu item is highlighted. Sub-items are selected in the same manner as regular menu

items. Any item may have a single letter command associated with it. That is, the item may be selected by pressing a command key and the designated letter simultaneously. When selected an item may be highlighted by a predefined image, by complementing, or by being boxed. A check mark may be placed (or removed from) in front of the text of an item when it is selected.

Programmer defined gadgets may be placed anywhere on a window. They have attributes of size, location, background colour, text colour, and border colour. They may be set as disabled (ghosted) or enabled. A predefined image may be associated with a gadget. Text (and its location) may also be assigned to a gadget. When a gadget is selected via the mouse it may be highlighted by complimenting, by being boxed, or with a previously defined image. A gadget may be set to automatically repeat without repeated clicks of the mouse. Or a gadget may have the toggle attribute where each time it is selected it is toggled on or off.

There are three types of gadgets: boolean, string, and proportional. A boolean gadget is selected on or off. A string gadget defines a protected area for a single string input. Associated with a string gadget is a fixed size buffer that may be larger than the protected area and may have initial text placed in it for modification. The user may edit the string in the gadget (which supplies a cursor) and signal completion by pressing the RETURN key. A proportional gadget is set within a defined frame. It is moveable by the mouse pointer in either or both the horizontal and vertical directions. Its size may be set as a proportion of some partially visible object like a list. Its location is returned as a proportion of the size of its frame both horizontally and vertically.

Two types of *requesters* are defined: alert and string. The system places a requester box in the centre of the screen and waits for one of the *buttons* (boolean gadgets) to be selected. An alert requester may have up to four lines of prompt text

and from one to three buttons. One of the buttons can be set as the default which gets selected by pressing the RETURN key.

A string requester prompts the user with up to five lines of text. A string gadget with a buffer of up to 80 characters is provided for a string response. There are two buttons, one marked "OK" and the other "CANCEL". The foreground and background colours, and the inverse attribute may be set separately for each line of prompt text.

A complete set of graphic tools are provided for drawing.

4. System Design

The focus of this thesis is on the courseware management and sequence control aspects of CAI. However, these aspects are described in the context of a complete multi-user CAI environment. This environment is built around the ERAS authoring language (section 2.2.1). This chapter describes the design of the over all system as well as the visual authoring environment for this system. Chapter 5 describes the computer prototypes which were developed to illustrate the courseware management and sequence control aspects of this system in the context of a single-user authoring environment.

The courseware management (section 4.3) and sequence control (section 4.4) aspects of the design were greatly influenced by the concepts of abstraction from computer science (section 3.1). The hierarchical structure of the courseware database and the modularity of the ERAS language allows for a "top-down" approach to design. "High level" abstractions can be created without the need to be concerned with implementation details. This structure also provides for the scoping and visibility rules for objects in the database.

Much of the design of the visual authoring environment (section 4.6) was inspired by some of the visual programming systems reviewed in section 3.2. In particular the Spatial Data Management System and the Program Visualization system, both developed at the Computer Corporation of America, influenced the design of the file card metaphor (section 4.6.1) and the tree structure editor (section 4.6.2) as means for accessing large databases and displaying the contents of these databases.

4.1 Desirable Characteristics

Listed below are seven characteristics that were felt to be desirable in the design of a modern computer-assisted instruction system.

Ease of Use - As the developers of Authorware (1989) state, there must be an effort to balance ease-of-use and power. This is reinforced by C.M. Brown's (1988) statement that "a critical step in defining the design philosophy for the user interface is to establish the appropriate balance of ease of learning, ease of use, and functionality" (p. 13).

Transportability of System and Courseware - This CAI system and associated courseware are designed to be implemented on the Elf (Educational Language Facility) system (see appendix A). One of the primary goals of the design of the Elf system is that it be easily transported to different computer hardware systems and operating systems.

Modularization of System and Courseware - By making the CAI system modular, new features may easily be added. For example, if an improved graphic display system becomes available, it could be substituted for the older system. Large scale CAI courseware projects should be easier to manage if developed in a modular fashion, particularly if many authors are involved. Modularity also supports the design level of courseware development by allowing a top-down development philosophy.

Hierarchical Courseware Database - This would support both the organization of courses that use a hierarchical structure and the management of courseware development teams by allocating different sub-trees of the course under development to each member of the authoring team.

Content Structure Reflected in Courseware - Often content is organized in a hierarchical structure. This can be reflected directly in the hierarchical structure of the courseware modules.

Generalizability of Interface - The user interface should have the same look and feel at all levels of system management and courseware development. For example, there should not be different rules for developing lessons, courses, or curricula.

Operating System Transparency - Except for systems programmers, none of the other users of the CAI system should need to know anything about the underlying operating system. In general, users should not be concerned about file formats, network protocols, or backup systems. The CAI system should look and feel the same to users whether it resides on a stand-alone microcomputer or on a large scale wide-area network.

4.2 The System Environment

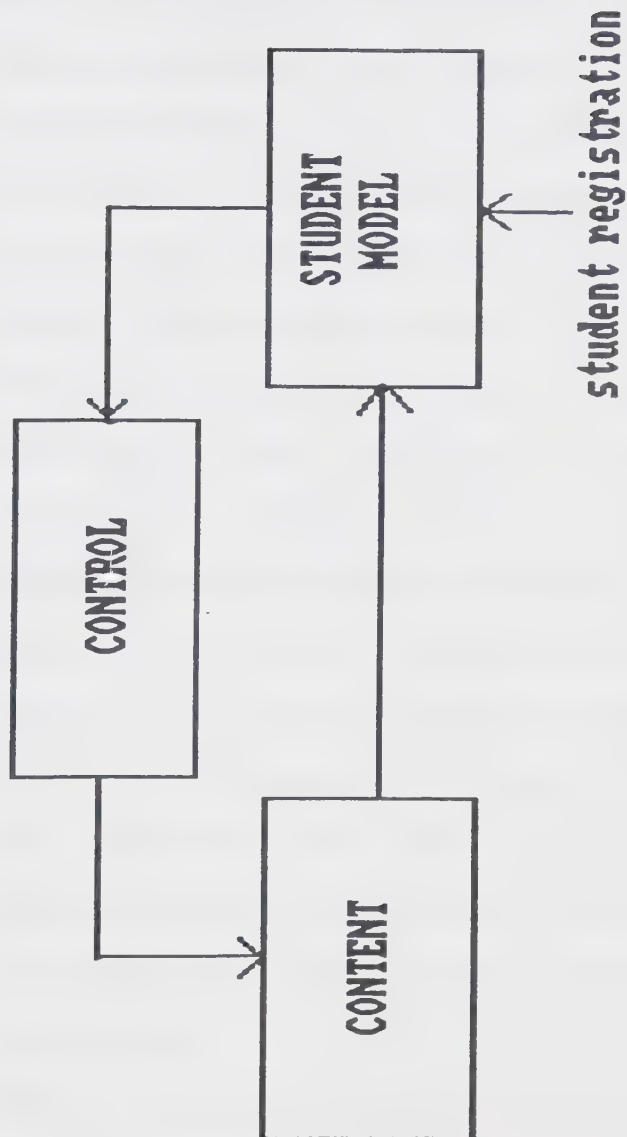
The CAI environment envisaged is one that separates the control and content components of courseware. It allows for the modification and monitoring of the execution environment by a student model component (see Figure 2). The major focus is on the control component and to a much lesser extent on the content component. The student model component is not included in the design.

This proposed CAI environment could be implemented on a wide range of systems, from a stand-alone microcomputer supporting a single author to a very large scale, wide-area network supporting a wide range of CAI users. The underlying CAI system must, however, be transparent to all users except systems programmers. That is, it should look and feel the same whether the user is at a microcomputer or a terminal attached to a wide-area network. Full implementation would require terminals with graphic user interfaces similar to those listed in section 3.4.

A complete CAI system should have the following subsystems:

1. Courseware development subsystem
2. Courseware presentation subsystem

Figure 2. Courseware Components



3. Examination and testing subsystem
4. Performance recording and analysis subsystem
5. Registration subsystem
6. Monitoring subsystem

The design which follows allows for the inclusion and support of all these CAI subsystems, however, these components are not all included in the design.

The system would support the following classes of users:

1. **Systems programmers** - They are required to install and maintain the CAI system on the underlying hardware and operating system. They would be responsible for the system's integrity including the backup of system software and courseware. They are the only users who would need to know the details of file and courseware storage.
2. **Managers** - They are chiefly concerned with the registration subsystem. They register other users on the system and assign system access privileges. They also oversee the general management of courseware and course data.
3. **Authors** - They are responsible for the development, evaluation, and maintenance of courseware and related course data.
4. **Instructors** - They supervise students taking CAI instruction. They may have registration privileges for student users and/or classes (groups of student users taking the same courses). Instructors generally work with managers in the registration of courseware for use by students. They also control student access privileges to course components for review, browsing, and examinations.
5. **Proctors** - They supervise students on the system and have access to the monitoring subsystem.
6. **Students** - They take instruction via selected system courseware.

On a stand-alone microcomputer system one person might fulfil the role of all the first five user categories. On large systems, individuals are likely to be assigned to only one of the six categories of users.

4.3 Courseware Management

The organization and storage of all components in this CAI environment would be in a multi-branched, hierarchical tree structured database. The use of such a database permits any node in the tree to be an abstraction of the sub-tree for which it is the root.

To a manager, this facilitates the control and management of the CAI system. For example, a university CAI system might allocate one branch of the site root to each faculty, each branch of each faculty node to a department within that faculty, and so on until individual courses were allocated. Large courses under development could be further broken down to the components for which each author was responsible. Production courses could be allocated to individual instructors for delivery to students.

To an author, it may be used as a tool at the design stage of courseware development. This will be illustrated in the design discussion below and in the implementation chapter.

4.3.1 LEVELs and ENTRY LEVELs

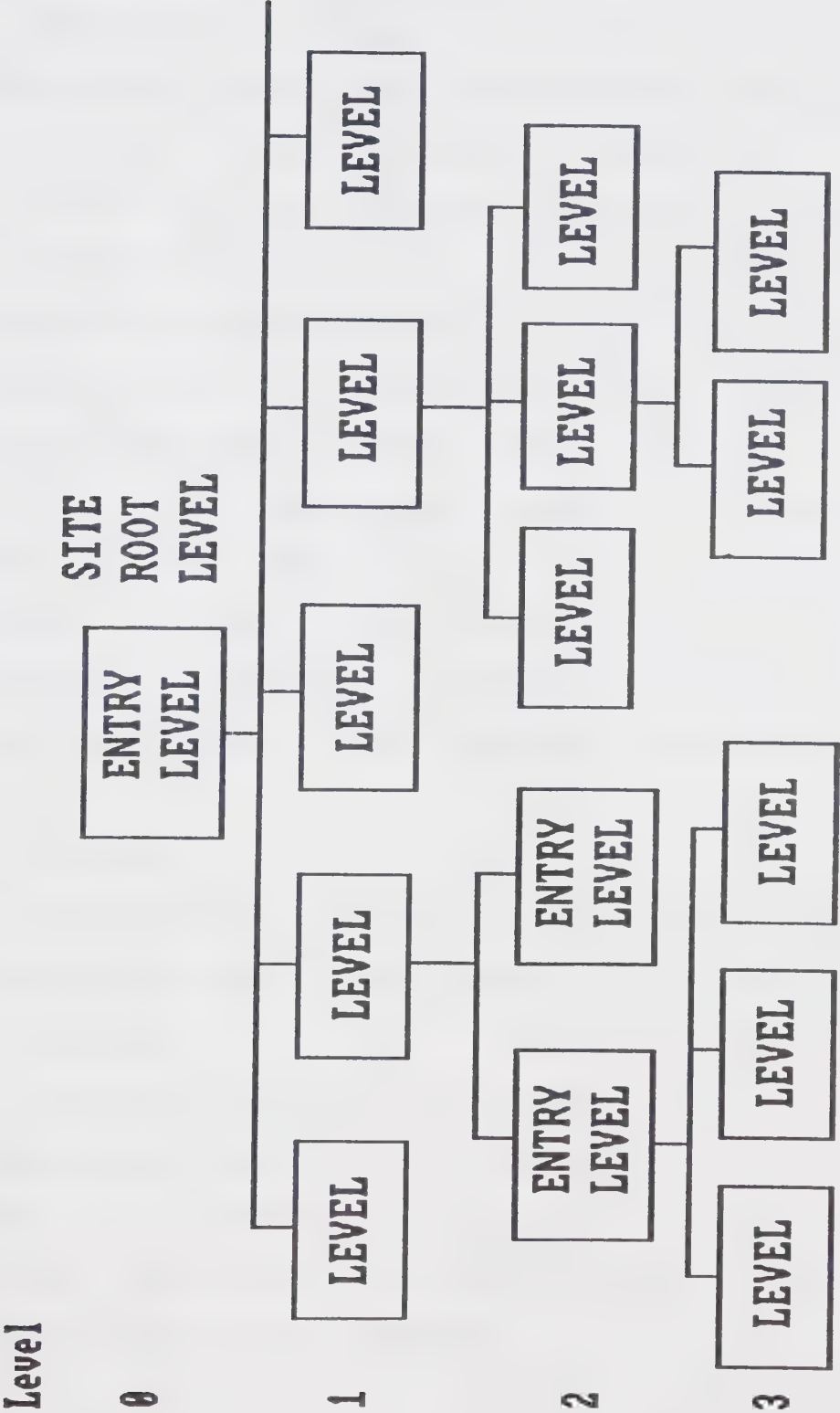
Each node in this tree is termed a LEVEL. The root node is called the SITE ROOT LEVEL. A site may range from a stand-alone microcomputer to a wide-area network. The SITE ROOT LEVELs for the two implementations discussed in chapter 5 are for a set of courses on (1) a multi-user minicomputer and (2) a stand-alone microcomputer.

Entry into this tree structure from the underlying computer system for the purposes of managing, editing, or executing components in the tree may only be done at specified LEVEL nodes termed ENTRY LEVELs (see Figure 3). That is, the registration subsystem of the CAI environment permits entry to the tree by specified registration privilege. For example, an author of a course would have entry and edit privileges at the ENTRY LEVEL of the course being created. This privilege then extends to all branches of the tree connected below that ENTRY LEVEL. A student would have entry and execution privileges at the ENTRY LEVEL of all courses registered to be taken. However, student access to some components may have special limitations (see section 4.3.5). In the above two examples, each ENTRY LEVEL would be the root of a course tree. A manager would have entry and management privileges at the ENTRY LEVEL for that part of the tree where that manager was responsible for registration of users and courseware.

Ownership of each ENTRY LEVEL and all attached lower LEVELs in the tree is designated to a specified user who is registered as a manager, an instructor, or an author. A manager-owner may assign ownership of any branch of the sub-tree by designating an owned LEVEL as an ENTRY LEVEL and registering a user (manager, instructor, or author) as that ENTRY LEVEL's owner. However, the manager retains senior ownership privilege and thus may deallocate ownership for that ENTRY LEVEL. There must be a site manager who owns the SITE ROOT LEVEL. Ownership also implies certain privileges (assigned at registration) which may include some, or all, of executing, creating, modifying, and deleting objects that exist at an owned LEVEL. The type and nature of these objects are discussed below.

This tree structure for the CAI environment can be very small or very large depending on the nature of the organization involved. On a stand-alone microcomputer there may only be one owner, the site manager; only one ENTRY LEVEL, the SITE ROOT LEVEL; and a simple CAI environment of a single lesson. At

Figure 3. LEVELs and ENTRY LEVELs



the other extreme, the CAI environment may encompass a wide-area network of many computers on a university campus where the tree structure nodes are organized by faculties, departments, curricula, and courses.

The implementations in chapter 5 have a SITE ROOT LEVEL and any number of "course" ENTRY LEVELs. There is a "course" registration system but no user registration system. All users have manager and author privileges.

An owner may designate a generic name for all LEVELs in the tree that exist at the same depth. Each individual LEVEL also has a *given* name. The ENTRY LEVEL generic name and given name are assigned at registration. An author might assign all branches from the ENTRY LEVEL the generic name "Chapter" and give each LEVEL at that depth a chapter title as a given name. It is also recognized that an owner may want to use more than one generic name at the same depth. For example the author may wish to have both "Chapter" and "Quiz" used as generic names at the same depth in the tree. "Quiz" would thus be termed an alias for "Chapter". This concept of generic names (and their aliases) and given names for each LEVEL is illustrated in Figures 4 and 5. This part of the design is included in the implementations covered in chapter 5.

4.3.2 Identifiers

Each LEVEL contains specialized directories to objects of various kinds that are required for the execution of courseware within the CAI environment. Each object has an identifier.

Identifiers are symbolic names for such objects as courses, course levels, modules, routines, functions, restart points, constants, types, and variables. The terms constant, type, and variable are used in a manner similar to their use in Pascal and Modula 2. (Syntax descriptions used in this thesis follow the Elf standard outlined in appendix A.) The syntax for an identifier is:

Figure 4. Example of Generic Level Names

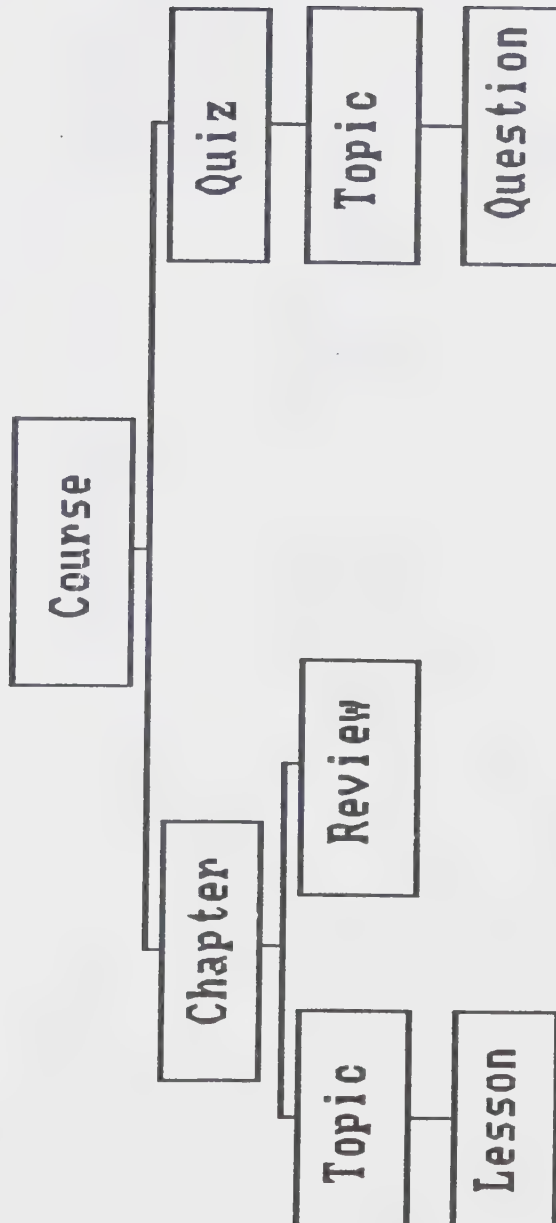
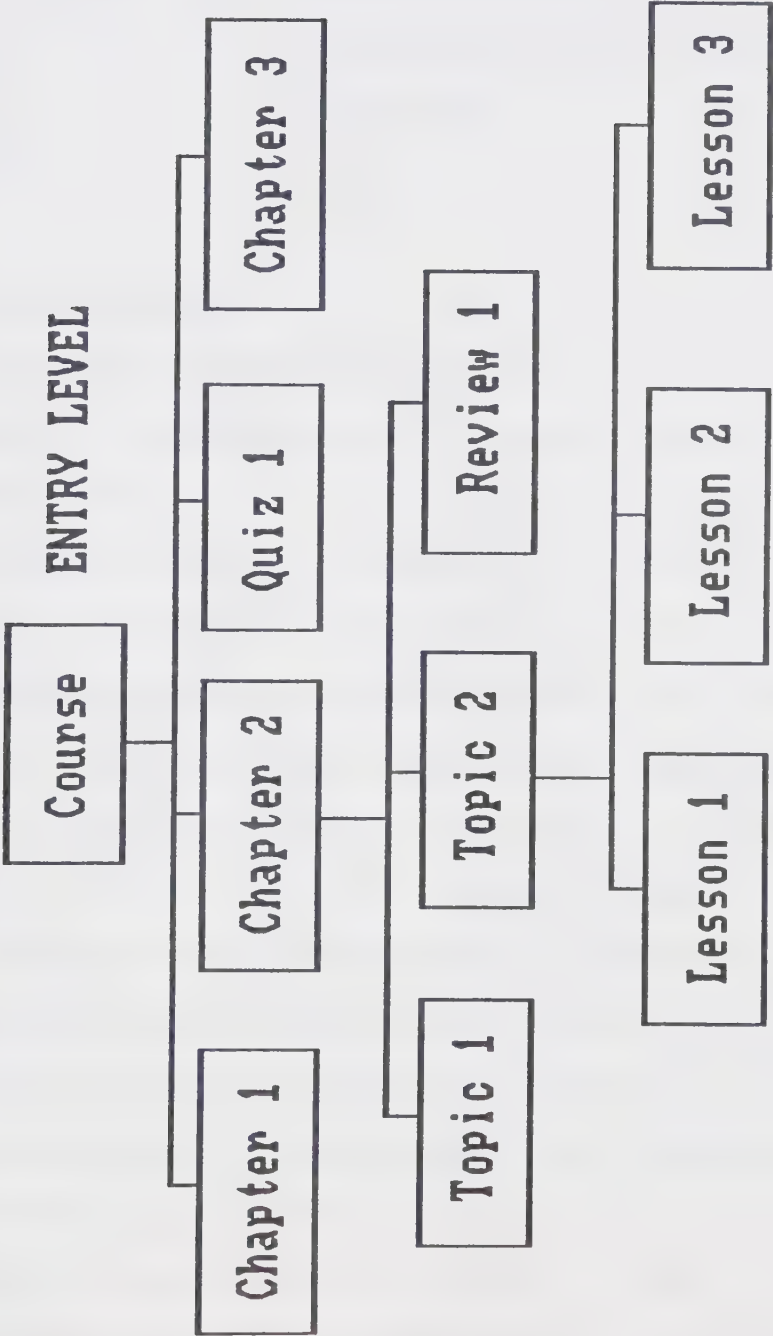


Figure 5. Example Course Tree



<identifier>

```

is      1 of      ("abcdefghijklnopqrstuvwxyz"
                  "ABCDEFGHJKLMNOPQRSTUVWXYZ"
                  "_#$%")
0 to 22 of      ("abcdefghijklnopqrstuvwxyz"
                 "ABCDEFGHJKLMNOPQRSTUVWXYZ"
                 "0123456789"
                 "_#$%");

```

4.3.3 Scope of Identifiers

The scope of identifiers is static and is similar to that used in Pascal and Modula 2. An identifier defined at one scope level is known at all lower scope levels but may be redefined at a lower level. For example, a NUMERIC variable, say, NumRight, when defined at one level, is known and may be referenced at that and all lower levels. However, at some lower level it may be redefined as, say, a LIST OF NUMERIC. Then the outer NumRight may not be referenced at that and lower levels because it has been replaced by the local NumRight. The outer NumRight still exists but may only be referenced when processing returns to the higher level. An object with an identifier that may be referenced is said to be visible.

Thus the tree structure determines the scope of all identifiers. To an author, all identifiers in the tree structure above that author's ENTRY LEVEL, for which access privilege (read and/or write) is permitted, are designated as at the system scope level. Care should be taken by users (with creation privilege) in redefining system constants, types, variables, routines, and functions as they would then lose access to them within their sub-tree. To assist inexperienced users, system identifiers could be protected from redefinition by the setting of a switch in the user's registration record.

4.3.4 Case Sensitivity of Identifiers

System level identifiers are case insensitive. When referenced, the author may type the alphabets in either case. When listed, all system identifiers are in upper case.

At the time a course is set up for an author it must be decided if the identifiers should be case sensitive or not. If they are case sensitive, whenever they are referenced the author must type the identifier in the correct case, that is, 'NumRight' would be a different identifier than 'Numright'. And, of course, they will be listed in upper and lower case.

However, if the author decides that identifiers will be case insensitive, then when referenced they may be typed in either or both cases. The author may decide at anytime how course identifiers should appear in listings (on the screen or in print). The choices are uppercase, lowercase, or as typed when defined. This choice may be made for all classes of identifiers or for specified classes. The defaults are: variables in lower case, types and constants in upper case, and all others as typed when defined.

For the prototype implementations all identifiers are case insensitive but are displayed as typed when defined.

4.3.5 Restart Points and Control Modes

A student may be in one of two modes: course control mode or student control mode. A *restart point* automatically generates either a *course control mode* or a *student control mode* restart record in the student's restart record file. This restart record contains all the information the CAI system needs to properly restart the student at that point of the course.

When in course control mode, the student's flow through the course is completely controlled by the control component and the student model component. If

the student interrupts this normal course flow to back up, review, or browse, then student control mode is entered. This may affect performance recordings, module access, and exam entry as determined by the instructor. If the student signs-off from the computer system in student control mode then the student control mode restart record will be used to restart the student at the next sign-on. When the student selects to leave student control mode the course control mode restart record is used for restart at the last restart point passed while in course control mode. If a student signs-off while in course control mode, the course control mode restart record is used to restart the student at the next sign-on.

An instructor may decide which, if any, of the student control sub-modes are available to the student. The back up sub-mode allows the student to back up to previous displays. The instructor may limit the number of previous displays through which the student may back up. Because of implementation restrictions, the system may also impose some upper limit as well. The review sub-mode allows the student to select for review any unlocked portion of the course that has been marked in the student's records as having been completed. The browse sub-mode permits the student to select for review **any** unlocked portion of the course. The instructor may lock portions of the course, such as exam modules, to these last two sub-modes. The instructor also has control over permitting access to certain of the browse sub-mode's features, such as access to correct answers, movement without responding to questions, etc.

Restart points are both implicit and explicit. Entry into any LEVEL is an implicit restart point. The name of that LEVEL is the identifier for that restart point. Within any control or routine module, an author may explicitly insert a restart point. The label for that explicit restart point is appended to the LEVEL name (with an intervening dot) to become the qualified identifier for that restart point.

Since only an author mode is implemented in the prototypes, restart points and control modes are not included in the implementations described in chapter 5.

4.4 Sequence Control

As specified in section 2.2 (Expectations), the execution of this CAI environment is based on six predefined languages and their interpreters. These are:

- o CONTROL
- o CONTENT
- o DISPLAY
- o INPUT
- o ANSWER
- o MENU

These languages were originally defined and implemented using the Elf System (appendix A) by Chiu (DISPLAY), Garraway (CONTROL and CONTENT), Higham (INPUT and MENU), and Nesbit (ANSWER) as part of the Educational Research Authoring System (ERAS) project under Hunka (1988a).

Since these languages were only available on the DEC VAX Elf System, they could only participate in the VAX implementation given in chapter 5 and not in the *AMIGA* implementation.

Objects, termed **MODULES**, written in one of these languages may exist in specialized directories (by module type) in each **LEVEL** node. For example, within a **LEVEL** is a **CONTENT DIRECTORY** containing **CONTENT MODULEs** written in the **CONTENT LANGUAGE** which may be executed by the **CONTENT INTERPRETER**. There is one exception; the **CONTROL LANGUAGE** may be used to write three types of modules: **CONTROL MODULEs**, **ROUTINE MODULEs**, and **FUNCTION MODULEs**. Each of these types of modules are executed by the **CONTROL INTERPRETER**.

The CONTROL LANGUAGE is a procedure type language which may be used to write algorithmic procedures and functions. It may be used to write support routines and functions as well as to control the order of selection of courseware content. Each instance of a LEVEL must have one and only one CONTROL MODULE. On invoking an instance of a LEVEL this CONTROL MODULE is executed. Any module written in the CONTROL LANGUAGE may call visible CONTENT MODULEs (Figure 6). A CONTROL MODULE may invoke an instance of a LEVEL at the next lower depth of the tree structure. When a module completes execution it returns control to the module that invoked it.

ROUTINE MODULEs may be called from any type of module as support routines. FUNCTION MODULEs are called from within expressions and return either a NUMERIC or a STRING result. ROUTINE MODULEs and FUNCTION MODULEs may be recursively entered and are created in the same way as CONTROL MODULEs.

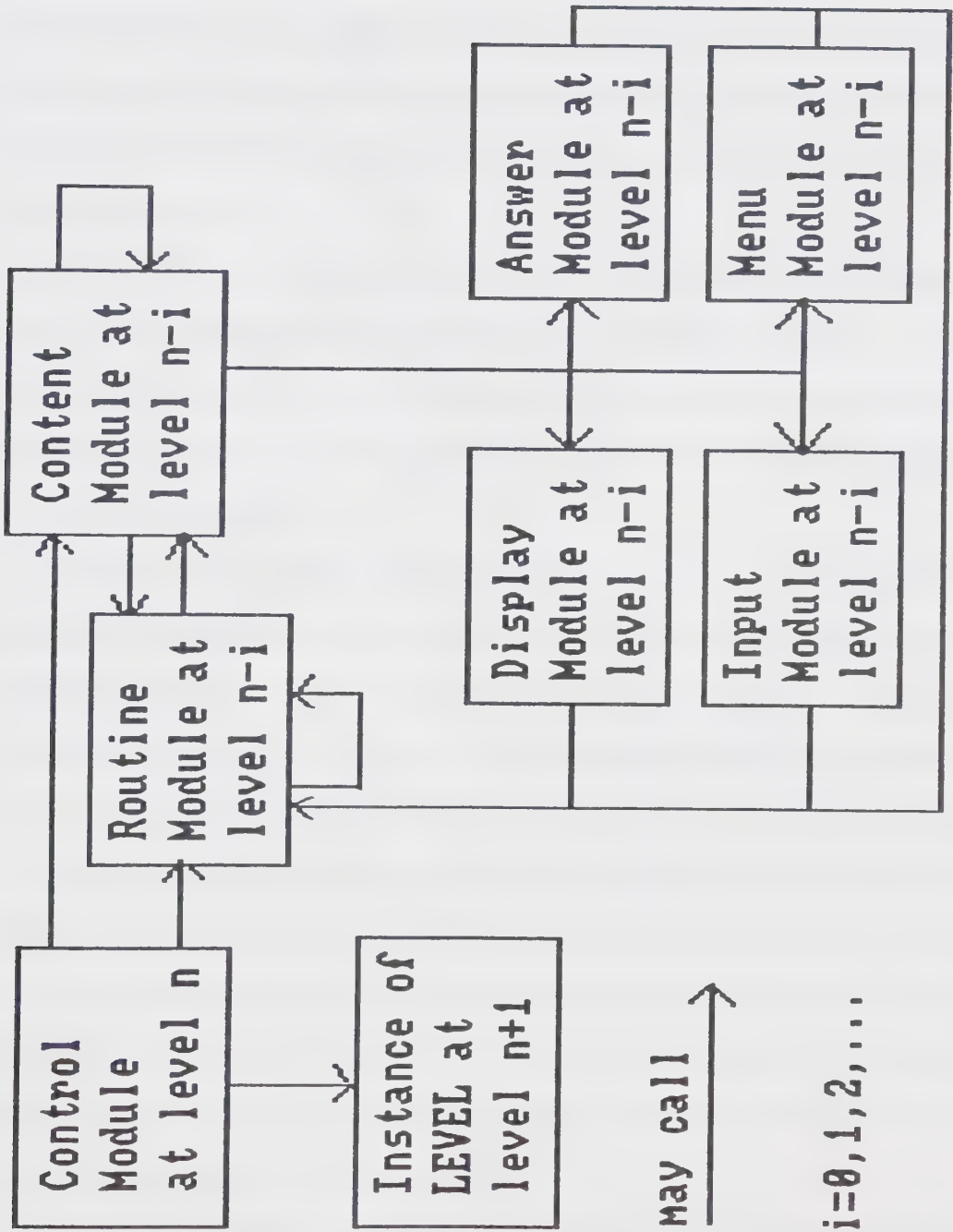
The following control abstractions are supported by the CONTROL LANGUAGE:

- o LOOP ... EXIT ... ENDLOOP
- o WHILE ... ENDWHILE
- o REPEAT ... UNTIL
- o FOR ... TO ... STEP ... ENDFOR
- o IF ... ELIF ... ELSE ... ENDIF
- o CASE ... WHEN ... OTHERWISE ... ENDCASE

A formal specification of the CONTROL LANGUAGE is given in appendix C.

The CONTENT LANGUAGE is provided for the structuring of CONTENT MODULEs. The presentation of information to the student and the processing of student responses to questions is handled by CONTENT MODULEs. A CONTENT MODULE presents the content of a small portion of a course in a linear sequence selected by the author. The elements in this sequence may be displays of text and graphics, the posing of questions, and the analysis of, classification of, and reprise

Figure 6. Module Invocation



to a student's response. Intermediate computations and the calling of subroutines may also be done within a CONTENT MODULE. CONTENT MODULEs may be invoked by other CONTENT MODULEs, and by CONTROL and ROUTINE MODULEs which control the general flow of a student through a course.

A CONTENT MODULE may call DISPLAY MODULEs, INPUT MODULEs, ANSWER MODULEs and MENU MODULEs. A CONTENT MODULE may also contain unnamed blocks of code written in the DISPLAY, INPUT, ANSWER, or MENU LANGUAGES. A formal specification of the CONTENT LANGUAGE is given in appendix D.

The ANSWER LANGUAGE, in addition to providing the means to judge a student's answer, also provides a special intrinsic branching structure that assists the author in providing control over feedback and other actions that are conditional on the student's response. A partial specification of the ANSWER LANGUAGE is provided in appendix E.

The DISPLAY LANGUAGE provides for the control of all text, graphic, animation, audio, and video presentations. The INPUT LANGUAGE provides for the control of student input from the keyboard and from pointing and selecting devices. The MENU LANGUAGE provides for special formatting and selecting features necessary for menu presentation and selection, and multiple choice type questions.

All MODULEs may take parameters of both call-by-value and call-by-reference. Formal parameters may provide for default values or default references. MODULEs may define local constants and variables, and may reference any visible routine, function, constant, type, or variable. The scope of local constants and variables is local to the MODULE and are unknown to locally referenced MODULEs unless passed to them by reference.

As was stated above, each instance of a LEVEL must contain one and only one CONTROL MODULE. In addition it may contain a number of directories to other

types of objects. One of these is the NEXT LEVEL DIRECTORY that contains entries that point to all LEVELs that branch from this node of the tree. All the other directories point to MODULEs or data objects that are defined for this LEVEL. The basic directories for modules are: CONTENT, DISPLAY, INPUT, ANSWER, MENU, ROUTINE, and FUNCTION. There are directories that point to definitions of TYPES, VARIABLES, and CONSTANTS. Other directories may be defined for data objects such as text, drawings, pictures, or graphs which could be referenced from DISPLAY MODULEs. The LEVEL may contain other objects that participate in the scope rules of the CAI environment, such as default display attributes or display screen definitions. The system may install other types of directories to support courseware development, such as video discs.

4.5 Computation

An important feature of the CAI environment, that is sometimes neglected, is provision of a powerful computation capability. This requires the ability to define data types, both simple and complex, and variables and constants of defined and predefined types. An expression evaluation language is also required.

The predefined computation aspect of the ERAS language was part of the DEC VAX implementation. It is described here as an example of what might be part of a complete CAI system. An alternate, more powerful, type specification is suggested in section 6.3.

The ERAS language supports two simple types: NUMERIC (three decimal place fixed point real) and STRING (maximum length of 32,767 characters). Two type constructors are supported: RECORD and LIST. A complete formal specification for type, variable, and constant declarations as well as a general description of the assignment statement and expression evaluator is given in appendix B. A formal specification of the assignment statement and expressions is given in appendix C.

Identifiers for types, variables, and constants may be declared at any point in the authoring process simply by the author requesting the declaration mode. This may be done automatically if the author references an identifier that is undeclared. Declarations at any scope level may be created, viewed, or modified. This was partially implemented in the DEC VAX prototype and not implemented at all on the *AMIGA* prototype since the ERAS interpreters were not available on that platform.

4.5.1 Variables and Constants

Variables are named references to locations in memory which may contain values pertaining to some type. These values may change during program execution.

Variables must be declared as belonging to some type before being referenced. Space allocation is done at execution time on entering the *MODULE* or *LEVEL* where the variable is declared. Variable space is deallocated on exit from the *MODULE* or *LEVEL* where it was declared. A variable may not change its type during execution.

Constants are named data items whose values are established at author time and cannot be changed at execution time.

4.6 Visual Authoring

Using a traditional CAI language like NATAL, the author would write a lesson in NATAL source code using a text editor, then compile the code with a NATAL compiler. If the compiler finds any syntax errors it is necessary for the author to return to the editor to correct the errors and then compile the code again. This process is repeated until all syntax errors have been removed. Next, the author must link runtime libraries to the compiled object code before viewing the results. If the lesson does not work as expected, again the author must return to the editor to make changes to the source code and start the process once more. When the lesson is

finished the text editor must again be used to edit a course map file that manages all the lessons in the author's course.

This tedious process is replaced in a visual authoring environment by a graphic interface that should represent the CAI system in a way that is meaningful to the non-programming author. This section describes, in a general way, the graphic interface needed for the CAI system specified above.

The primary user interface is represented by a file card metaphor to portray the CAI system. This is supplemented by five other graphic interfaces that perform special duties. The first of these is used to traverse and manipulate the tree structure of the courseware management aspect of the system. The second is used to represent and edit the Pascal-like control structures of the CONTROL LANGUAGE and the statements of the other languages. The third graphic interface is used to display and build numeric and string expressions, and assignment statements. The fourth interface is used to build the presentation displays that would be seen by the student. The last interface is used by systems programmers to examine and maintain the objects of the system which are represented in the traditional desk top metaphor.

In all of the above graphic interfaces, objects may be selected for some action. The available actions can be accessed from pull down menus or selection buttons. The more frequently used actions would be available via function keys and/or by pressing a command key plus a single mnemonic key. Those actions available at all times would be in the left most menus. Listed items that are not available would be *ghosted*. If a menu item has a keyboard equivalent, this would be displayed in the pull down menu.

Error conditions are reported to the user in a manner that is appropriate to the type of user. For example, the same type of run-time error would be reported differently to an author and a student. To the author, the error report would provide advice and information to correct the error. To the student, it would provide advice

on how to proceed if possible or who to advise if unable to proceed. All errors are recorded in an error report file for later reference.

There is a designated *HELP* key. This must be the labeled *HELP* key if one exists. The equivalent *HELP* function is always available from the left most pull down menu. Help information is context sensitive.

4.6.1 File Card Metaphor

In this system colour is used to highlight titles, choices, and selection buttons. Each file card displays relevant data needed by the author plus labeled selection buttons. Buttons that are inactive are *ghosted*. An inactive documentation or directory button indicates that there is no documentation or the directory is empty. The "Create" drop down menu allows for these items to be created. Once created the corresponding button becomes active.

When an author signs-on to the computer system the author's ENTRY LEVEL file card is presented. This card allows access to general course documentation, generic name editing, and the course root level.

Access to the documentation editor is obtained by selecting the "Documentation" button. Selecting the "Generic Names" button allows the author to create, view, or edit the generic names for the course LEVELs. By selecting the "ROOT LEVEL" button the author goes directly to the LEVEL file card for the ROOT LEVEL of the courseware being authored. However, menu selection allows direct entry to the tree structure interface and access to all LEVELs of the course. The ROOT LEVEL file card replaces the ENTRY LEVEL file card.

A LEVEL file card gives access to all directories for that LEVEL plus internal documentation, display and screen setting defaults (used by the DISPLAY LANGUAGE), and the CONTROL MODULE for that LEVEL. Selecting the

"Control" button, allows the author to edit the control module's statements, formal parameters, local constants, local variables, or documentation.

From the LEVEL card any of the directories for that LEVEL may be selected. The buttons for empty directories are *ghosted*. The "Next Level" directory is for all LEVEL's that branch from this LEVEL. Selecting one of these would place a new LEVEL file card on top of the present one but it would be one row down and one column to the right (cascaded) so as to expose the hierarchical path to that LEVEL.

A directory may list its contents in either alphabetical order or *logical* order (an order decided upon by the author). The choice is made by a menu selection. Items may only be selected from the alphabetic list directory. However, the logic list directory is more powerful in that it also allows the changing of the logical order of the items, the insertion of new items, and the deletion or renaming of items. Items may be selected for editing, or to be copied or moved to a transfer buffer to be pasted in a directory of the same type at another LEVEL.

Selecting an item from the CONTENT DIRECTORY, for example, would allow the author to edit that content module's statements, formal parameters, local constants, local variables, or documentation.

One of the pull-down menus is the *EXIT* menu. It is used to leave the current file card by selecting one of the items in that menu. The items indicate to what file card exit is desired. The choices are current module, current directory, current level, back one level, root level, entry level, and system. Inappropriate choices are *ghosted*. If changes have been made to the current file card, a requester pops up asking to confirm that the changes are to be recorded. It is also possible to recover previous versions of the file card. The number of previous versions that are retained is set at registration and may be limited by the particular implementation. Using the mouse, the author may also point to and click on visible LEVEL file cards to move directly to the selected LEVEL.

The file-card metaphor for courseware management was implemented in both prototypes. These are described, with screen representations, in the next chapter.

4.6.2 Tree Structure Editor

Although the file-card editor presents graphically the path to the current LEVEL via the cascaded display of the file-cards for all previous LEVELs, this does not show the relationship of the current node (LEVEL) in the tree to the nodes in the immediate vicinity. This is the role of the tree structure editor.

In this editor the current node is displayed in the centre of the screen as a node label. A node label is presented graphically in a form similar to an address label. Each node label has three lines which show the generic name and logical order number of the node, the given name for the node, and the version number and the date of the last modification of the node.

Immediately above and below the label for the current node are displayed the labels of its sister nodes, if any, in their logical order with respect to the current node. To the right of the label for the current node is a scrollable table of node labels of all the daughter nodes to the current node. By menu selection this table may list the daughter nodes in either alphabetical order or logical order. Immediately to the left of the label for the current node is displayed the label for its parent node, unless the current node is the root node. Above and below the parent node are displayed the labels of its sister nodes, if any, in their logical order with respect to the parent node.

Any visible node may be highlighted for some action by using the mouse to point to and click on the node's label. The 'Cancel' action removes the highlighting of the clicked on label and re-highlights the previously highlighted label. The 'Select' action brings the node to the centre of the screen, making it the current node. This causes an immediate rearrangement of the labels on the screen. The 'Enter' action makes

the selected node the current node, switches the display to the file-card editor which presents the node's file-card in its path's cascaded display.

Buttons are provided to scroll the table of daughter node labels up or down one node at a time, to page up or page down a screen length of the table, or to move directly to the top or the bottom of the table.

In addition to the 'Cancel', 'Select', and 'Enter' actions, other action buttons are available for the daughter nodes. The 'Delete' action causes a requester to appear to confirm or cancel the deletion of the highlighted node. The 'Delete' button is ghosted if the selected node is not a leaf node. The 'Move' action puts the highlighted node in a transfer buffer so that it may be inserted elsewhere in the tree. The transfer buffer is maintained between authoring sessions. The 'Move' button is ghosted if the transfer buffer is occupied. The 'Insert' action allows the author to insert before the selected node either the node in the transfer buffer or a new node. If this is a new node a requester asks for the generic name and the given name for the new node. The table of daughter nodes always ends with a dummy label marking the end of the table. This may be selected for the 'Insert' action. The 'Move' and 'Insert' actions are only available when the table lists the daughter nodes in logical order.

By menu selection the author may move between the file-card editor and the tree structure editor. The current node determines the display presented after the move from one editor to the other.

The tree structure editor was implemented in the *AMIGA* prototype and is described, with screen representations, in the next chapter.

4.6.3 Other Editors

When an author selects to edit the statements of a module, the control structure editor is invoked. This editor allows the author to insert, delete, move, copy, and

modify statements. It permits the author to view statements using both graphic (icon) type models and text models. The author may also restrict the depth of control structure nesting to be displayed. The amount of information displayed for each statement may be restricted so that more statements may be displayed on one screen.

Any statement may be selected for expansion. For example, if the author selected (or created) a CONTENT CALL statement a MODULE CALL requester file card would pop up displaying (or requesting) the name of the module to be called and a selector for the actual parameters.

The expression editor is invoked when an assignment statement is selected, or whenever an expression is needed such as for an actual parameter. This editor permits the creation and modification of expressions in standard algebraic form. This includes the use of terms and factors made up of complex numerators and denominators.

Using the pull down menus an author may request to search for visible variables, constants, or functions. When an expression is parsed any unknown variables are highlighted and the author is permitted to create the variable as a local variable or at some visible LEVEL of the tree. The use of a function name invokes a FUNCTION CALL file card similar to a MODULE CALL file card.

The display editor allows the author to interactively build the student display in an environment similar to those used for page layouts in desktop publishing software. This editor would have access to graphic, video, sound, and animation editors.

The control structure, expression, and display editors are not part of the prototypes developed for this thesis research.

The systems editor is used by systems programmers to examine and maintain the objects of the CAI system which are represented as icons in the traditional desk top metaphor. This editor may not be used to create objects but just examine them. It does, however, permit the systems programmer access to documentation and text

editors. This editor was developed in the *AMIGA* prototype and is described in the next chapter.

5. Prototype Implementations

5.1 The Elf System and the Commodore *AMIGA*

To demonstrate the feasibility of the design described in the previous chapter, two implementations of part of the courseware management and sequence control aspects of the CAI system have been completed.

The initial work was done on a Digital Equipment (DEC) VAX 11/780 super-minicomputer using the Elf (Educational language facility) system development environment. Elf was under development at the Division of Educational Research Services, Faculty of Education, University of Alberta. A brief description of the Elf System is given in Appendix A.

Interpreters for the six languages that constitute the Educational Research Authoring System (ERAS) (Hunka, 1988a) were fully implemented in Elf and formed the basis for the prototype implementation described below. (The Elf Language is used to describe the syntax of the ERAS CONTROL, CONTENT, and ANSWER languages in appendices B to E.)

The internal control structures for the CAI system were completed in Elf. Preliminary authoring and run-time environments were also implemented in Elf. These are described in the sections 5.2 to 5.5. Unfortunately, development work on Elf was interrupted, because of financial restraints at the University of Alberta, before the Elf graphic user interface could be fully implemented. This allowed only the file card interface to be attempted in Elf. The Elf System and all work developed on it became unavailable for further research when the DEC VAX of the Division of Educational Research Services was permanently shut down. The file card interface was only partially completed before the shut down of the VAX computer. Examples of the Elf interface, with descriptions, are presented in sections 5.6 and 5.7.

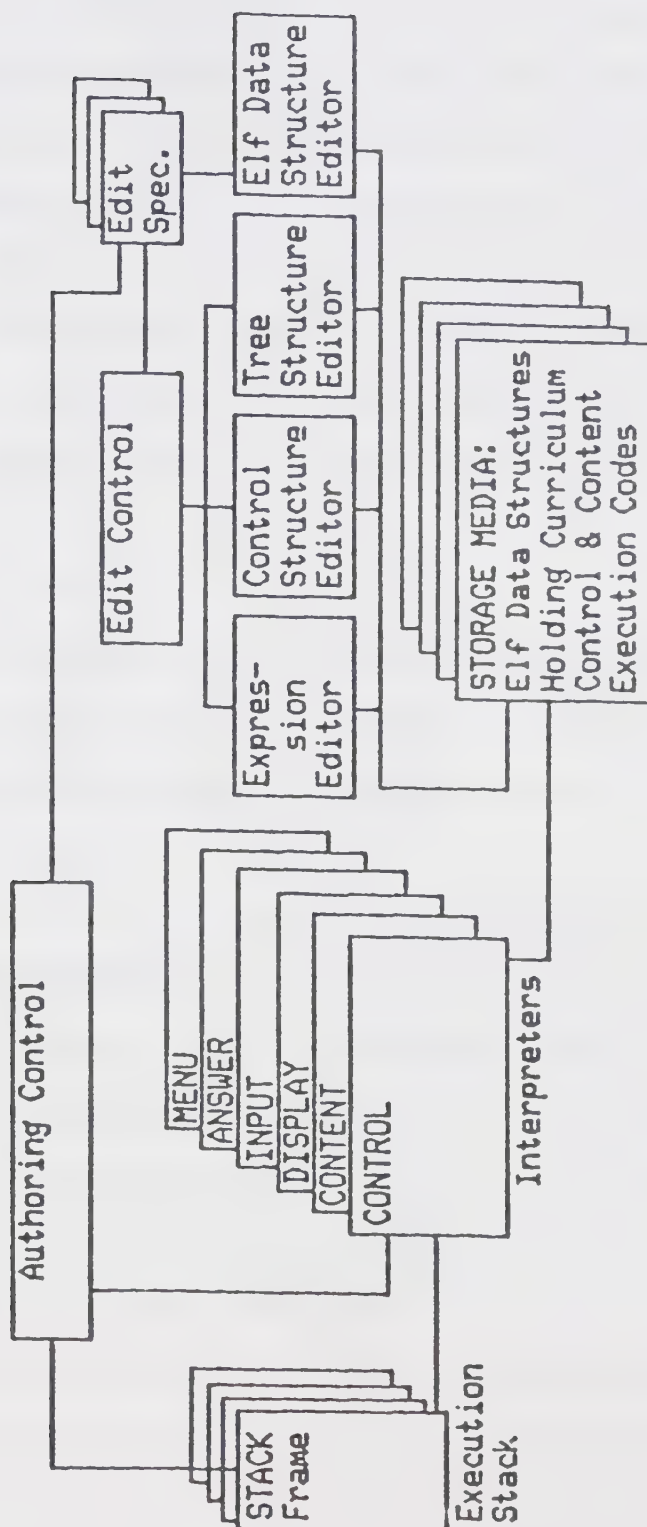
A more complete demonstration of the desired interface was developed on a Commodore *AMIGA* 500 with an A590 hard drive and one megabyte of memory. This prototype made use of the features of the *AMIGA*'s graphic user interface, Intuition, described in section 3.4. Programming was done in GFA Structured BASIC, version 3.5 (Engels, Görgens, & Ostrowski, 1988) with extensions from Extend, version 1.3 (Highway, 1989). GFA BASIC with Extend provided the necessary tools for the easy building of the interface prototype. Examples of the *AMIGA* interface, with descriptions, are presented in sections 5.8 to 5.12. The hierarchical directory structure of the *AMIGA* operating system was used to emulate the Elf *media* (see section 5.2) and data structures used in the original implementation of the CAI system.

5.2 Courseware Storage

Figure 7 is a diagram of the logical structure of the visual authoring environment as developed in Elf. The editors are used to build, display and modify the Elf data structures that hold the control and content components of the courseware. The interpreters for the CAI languages are used to translate the information in these same data structures to present the curriculum materials to the author as a student would see them.

Elf data structures are used to describe and store all elements of the CAI system and courseware. These data structures are kept in special Elf files called *media*. *Media* are dynamically created in memory and saved as relocatable blocks of memory. Data structures in a *media* are linked logically to each other by *abstraction*, an Elf device similar to a pointer. Whereas a pointer in a language like Modula 2 contains a memory address, an Elf *abstraction* contains information to locate a data object in a *media*. Each *media* has a root structure that is abstracted as the *entry abstraction*

Figure 7. Visual Authoring Environment



whenever a *media* file is opened and read into memory. This root structure is used to access all other structures in the *media*, either directly or indirectly, by *abstraction*.

The STORAGE MEDIA in Figure 7 are the Elf *media* that hold the CAI system's tree structured database and related courseware data. Every sub-tree in this CAI system associated with an ENTRY LEVEL is stored in its own Elf *media* file. There is at least one such *media*, the *site root media*, that contains the SITE ROOT LEVEL. A branch of the tree that points to an ENTRY LEVEL actually points to the *media* file for that sub-tree. The size limit of an Elf *media* is implementation dependent.

All curriculum information needed for the control and content of a course is stored in Elf data structures which, in turn, are stored in Elf *media*.

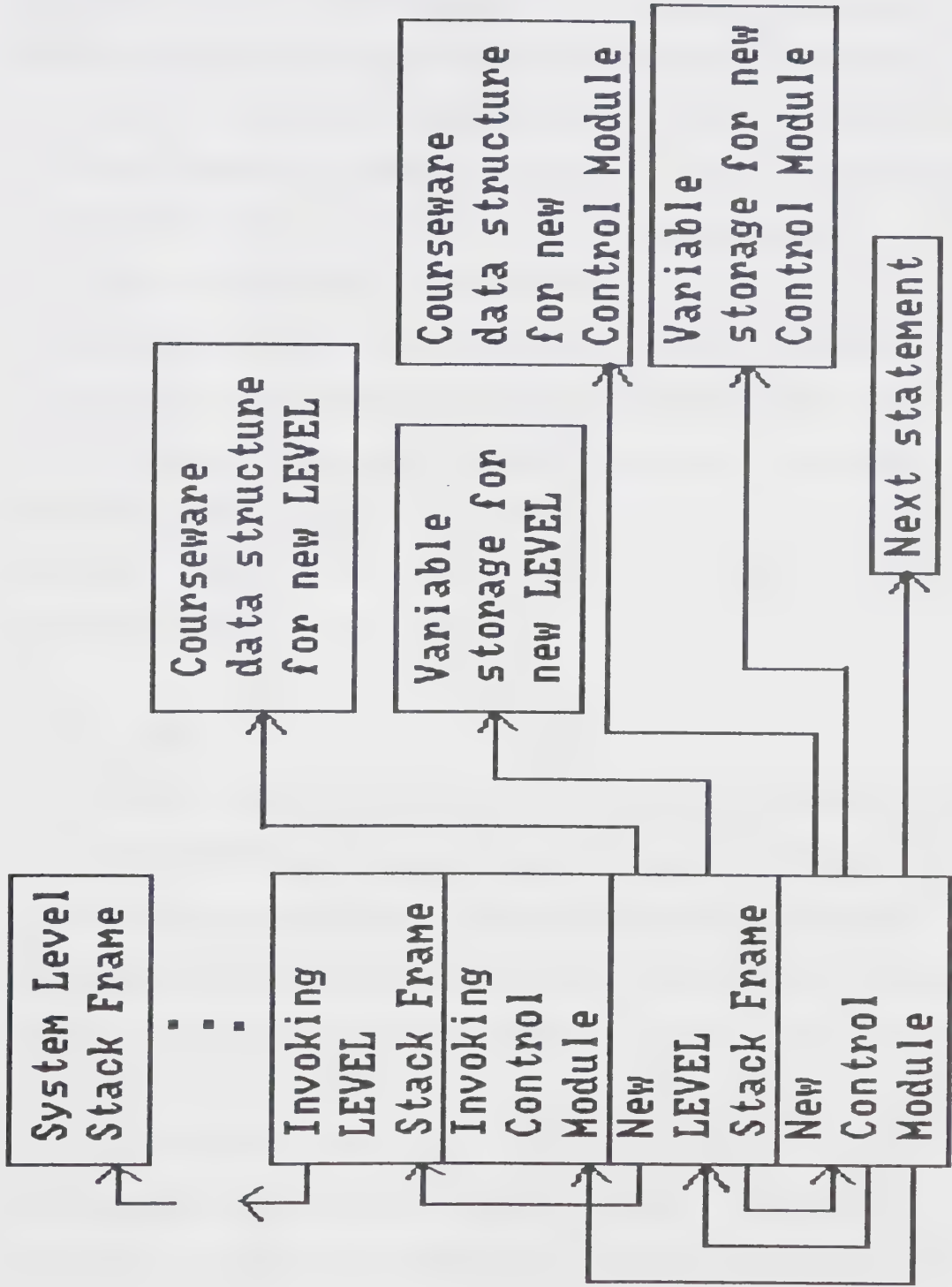
5.3 The Run-time System

The run-time environment is stack based. The execution stack frames (Figure 7) are Elf data structures in an Elf *media* and are dynamically linked by Elf *abstraction*. Saving this *media* completely preserves the run-time environment. It is this *media* that is placed in the student's restart record file.

On entering a LEVEL, a LEVEL stack frame and a module stack frame (for that LEVEL's CONTROL MODULE) are *pushed* onto the execution stack (Figure 8). The LEVEL stack frame abstracts each of the following data structures: (1) the stack frame of the invoking LEVEL, (2) the LEVEL's own courseware data structure, (3) the run-time storage of its variables, and (4) its CONTROL MODULE's stack frame.

When a module of any type is invoked, a module stack frame is *pushed* onto the stack. A module stack frame abstracts each of the following data structures: (1) the module stack frame of the invoking module, (2) the module's own courseware data structure, (3) the run-time storage of its local variables, (4) the data structure of the

Figure 8. Execution Stack



next statement of this module to be executed, and (5) the LEVEL stack frame for the LEVEL to which it belongs.

This stack system provides control over the scope and visibility of all identifiers. For example, when a variable is referenced it is first searched for in the current module's stack frame. If not found, the LEVEL stack frame abstracted by that module's stack frame is used to continue the search. If not there, the LEVEL stack frame one level above is invoked, and so on, until the system LEVEL stack frame is reached. If the identifier is not located there an error condition is raised.

After a module stack frame has been pushed on to the stack, its parameters and local variables are initialised and the appropriate ERAS language interpreter is invoked to interpret the execution codes that are stored in its courseware data structures. Once all its *statements* have been executed, which could have included calls to other LEVELs and modules, its stack frame is *popped* from the stack. If the module was a CONTROL MODULE then its LEVEL stack frame is also *popped*. Control is then returned to the module that invoked it.

5.4 The Editors

The Elf system provides a special editor for *media* files. It permits the creation, deletion, and modification of the data structures in a *media*. To use this editor a systems programmer, using the Elf edit specification language, creates an edit specification that describes the data structures to be edited, the method of displaying and modifying these structures, the organization of the *media*, and the interpretation of the keyboard.

Different edit specifications may be made for the same type of *media*, thus providing different views of the same *media*. For example, one view might be simplified for a beginning author while another may be quite complex for a systems programmer debugging the *media*'s organization.

The file card metaphor in the Elf prototype is implemented by one of these edit specifications. Every file card in the file card metaphor represents externally the contents of an internal data structure. For example, when an author selects "Control" from a LEVEL file card, a CONTROL MODULE file card appears. This indicates that internally an Elf data structure that represents a LEVEL abstracts a data structure that represents a CONTROL MODULE.

The Elf version of the file card metaphor was partially completed and is described in section 5.7. A final version of the file card metaphor was developed on the *AMIGA* and is described in section 5.10.

The expression editor and the control structure editor, which included the display editor, were not part of this thesis research. The tree structure editor and the systems editor were not completed as part of the Elf prototype but were developed as part of the *AMIGA* prototype (see sections 5.11 and 5.12).

5.5 The Authoring Environment

During an authoring session the run-time execution stack is always maintained down to the current LEVEL of editing or execution. Thus an author may switch freely between editing and executing the course. When executing a course, the author may interrupt execution, move up or down the stack and select to edit any MODULE or LEVEL that is reached. Execution resumes at the MODULE or LEVEL that was edited.

When editing a course the author may select to execute the MODULE or LEVEL being edited knowing that it will be executed in the run-time environment that would exist under normal execution.

This method also permits the checking for the existence of referenced identifiers. For example, an author creates a CONTENT CALL statement in a CONTROL MODULE. A requester file card pops up requesting the name of the

CONTENT MODULE to be called. The stack can be used to display the names of visible modules, or a name can be typed in and then searched for in the stack. If it is found, the author is informed of its generic and given names and asked to confirm that this is the required module. If it is, that module's use count is incremented and the author is prompted for any actual parameters needed to match the called module's formal parameters.

If the desired module is not found, the author is prompted to create an empty module in a CONTENT DIRECTORY within the present scope. This module's location is stored in a reminder file for the author to later reference as a module that must be completed.

Every time an object is referenced its use count is incremented. Whenever it is de-referenced, i.e. the calling statement is deleted, its use count is decremented. Only when its use count is zero is an author allowed to delete the object. This prevents dangling references.

5.6 The Elf Interface

As was mentioned in section 5.1 above, the first attempt to implement the file card metaphor for the CAI system used an early version of the Elf graphic user interface. This implementation verified the internal control design of the authoring environment as described above but could not incorporate all the desired features available in the more popular desktop type graphic user interfaces. For completeness, the original file card metaphor, as implemented in Elf, is described in this section and the following section. The full courseware management aspect of the CAI system's interface design was developed on the *AMIGA* and is described in sections 5.8 to 5.11.

The Elf graphic user interface had a multi-level design and was intended to be terminal independent. It was first implemented for the DEC Gigi colour graphics

terminal which at the time was the only available terminal on the Division of Educational Research Services' VAX computer. This terminal could interpret graphic commands sent from the host VAX 11/780 mini-computer via serial (RS232) connection. It could display structured graphics in eight colours. The Gigi terminal did not, however, support a pointing device such as a mouse.

Each file card of the file card metaphor represented an Elf data structure and was displayed as a rectangular box of some fixed size. If a file card had a subordinate relationship to an underlying file card, then this relationship was shown by having the subordinate file card displayed completely inside the parent's file card box. Headings, variable data and *SELECT* buttons could be displayed in a file card. Colour was used to highlight file cards of different types and to show the relationships among headings within a file card. Variable data were always displayed in one colour and *SELECT* buttons in another. However, colour information was redundant.

Variable data were of two classes. System variable data was set by the system and could not be altered by the author. Examples of system data are version numbers, creation dates, and modification dates. Author variable data could be altered by the author. Identifiers, comment text, and numeric expressions are examples of author data.

The Gigi's numeric keypad was interpreted by the Elf *media* editor as a set of edit function keys. These were used to move a cursor from object to object on the current file card and to create, select, or delete objects. The destination objects were either author variable data fields or *SELECT* buttons. If the ENTER key was pressed while the cursor was on a *SELECT* button, the object named in the button's heading was selected. The object was, in fact, an Elf abstraction "pointing" to another Elf data structure. If the data structure existed then a file card that represented that data structure would be displayed and that file card would become

the current file card. If the data structure did not exist then it was first created, attached to the current data structure by Elf abstraction, and then a file card for the data structure was displayed. Two keys could be used to leave a file card and return to the parent file card. The EXIT key was used if any modifications to the current file card were to be retained, while the QUIT key was used to leave the file card unaltered.

5.7 The Elf File card Metaphor

When an author entered an ENTRY LEVEL the Elf *media* for that entry level was loaded into memory, the screen was erased, and a *media* file card (Screen 1) was presented. This file card displayed the *media* name, version number, creation date, and the date of the last modification. All of these were set by the system.

SELECT buttons were available to move to other file cards. Three of these buttons would select a small text editor file card for internal documentation of the author's name and address or for a comment on the purpose of this entry level. An author's address file card is shown in Screen 2.

By selecting "Generic Names" the author could create and modify the generic names, or their aliases, for each level within the subtree controlled by this ENTRY LEVEL (Screen 3). Each "Generic_level" file card permitted the author to edit the generic name, select any number of aliases for the generic name, make a comment about that level of the tree, or select the "Generic_level" file card for the next level. Screen 3 shows a cascade of the first four "Generic_level" file cards inside the *Media* file card.

From the *Media* file card the author could select to enter the "Entry LEVEL". This produced a LEVEL file card one row down and one column to the right (Screen 4). This was the root level for the ENTRY LEVEL's subtree. (In the AMIGA implementation there was a change in nomenclature. The *Media* file card was renamed

Screen 1. Elf Media File card

Media Name: TOMS	Version Number: 1
Entry LEVEL: /SELECT /	
Creation Date: Feb 20 1987	
Modification Date: June 04 1987	
Author(s): /SELECT /	Author's address: /SELECT /
Media documentation: /SELECT /	
Generic Names: /SELECT /	

Screen 2. Internal Documentation Editor

Media Name: TQMS		Version Number: 1
Entry LEVEL: /SELECT		
Creation Date: Feb 20 1987		
Modification Date: June 04 1987		
Author(s): /SELECT	Author's address:	/SELECT
Media documentation: /SELECT		
Generic Names: /SELECT		
Division of Educational Research Services Faculty of Education The University of Alberta Edmonton, AB T6G 2G5		

Screen 3. Generic Level File cards

Media Name: TOPS		Version Number: 1	
Generic_Level Name: Course			
Generic_Level Name: Chapter			
Generic_Level Name: Topic			
Generic_Level Name: Lesson			
Alias :		<input type="text" value="/SELECT"/>	
Next Level:		<input type="text" value="/SELECT"/>	
Comment:		<input type="text" value="/SELECT"/>	

Screen 4. LEVEL File card

Media Name: TOMS		Version Number: 1	
LEVEL -- Course		TOMS	
Control: /SELECT /	Display: /SELECT /	Screen: /SELECT /	
Directories -			
Next Level /SELECT /			
Modules:	Data:	Support:	
Content /SELECT /	Types /SELECT /	Routine	/SELECT /
Display /SELECT /	Variables /SELECT /	Texts	/SELECT /
Input /SELECT /	Constants /SELECT /	Drawings	/SELECT /
Answer /SELECT /	Functions /SELECT /	Graphs	/SELECT /
Documentation: /SELECT /			

the ENTRY LEVEL file card and the "Entry LEVEL" button was renamed the "ROOT LEVEL" button.) The LEVEL file card gave the author access to all data structures abstracted by an instance of a LEVEL data structure.

The "Next Level" button selected a directory list of all daughter LEVELs of the current LEVEL. If one of these daughter LEVELs was selected its LEVEL file card would be displayed one row down and one column to the right in a cascade fashion similar to that shown in Screen 3. In this way the author could descend the various branches of the tree structure.

A LEVEL file card displayed the LEVEL's generic and given names and *SELECT* buttons for all directories attached to the LEVEL, the CONTROL module for the LEVEL, display and screen default settings, and LEVEL documentation text. There were four classes of directories. The "Next Level" class had only one member and this directory was discussed above. The "Modules" class had four directories listing CONTENT modules, DISPLAY modules, INPUT modules, and ANSWER modules. The "Data" class had four directories listing type definitions, variable declarations, constant definitions, and FUNCTION modules. The "Support" class had four directories listing ROUTINE modules, text files, drawing definitions, and graph definitions. From the LEVEL file card various objects could be selected for creation, examination, modification, or deletion.

5.8 The AMIGA Interface

In the AMIGA prototype the CAI system is referred to by the name of the set of original languages underlying the design: ERAS - Educational Research Authoring System. The complete courseware management aspect of the CAI system's interface design was implemented. This included the file card editor, the tree structure editor, and the system editor. There is no user registration subsystem. All users have system, manager, and author privileges.

As in the Elf prototype colour is used to highlight information, but is redundant. Details of colour use will be provided in the descriptions of each editor. Each editor uses a different background colour: the file card editor uses light grey, the tree structure editor uses tan, and the system editor uses light blue. The use of light background colours, with strong contrasting colours for text, follows the recommendations of C.M. Brown (1988). These three particular background colours were selected on the basis of their observed popularity on other software products widely used on the *AMIGA*. Many of today's software products allow the user to adjust the colour palette to the user's own taste and this feature could be added to this implementation.

The SITE ROOT LEVEL is named ERAS (Screen 5) and can have any number of daughter ENTRY LEVELs. These ENTRY LEVELs have the generic name "Course". On entry to the SITE ROOT LEVEL the ERAS system file card is displayed. This file card displays the following icons which may be selected by "double clicking" on them with the mouse pointer:

- | | |
|--------------|--|
| Register - | Provides manager access to the ENTRY LEVEL registration subsystem. |
| ERASAuthor - | Provides author access to all ENTRY LEVELs via the file card editor and the tree structure editor. |
| Courses - | Provides systems programmer access to all ENTRY LEVELS via the system editor. |
| Icons - | Provides systems programmer access to system editor icon designs and the icon editor. |
| GFABasic - | Provides systems programmer access to the GFABasic/Extend program development environment. |



Icons



DVI-DVI



Courses



Register - GEA



EASOn line - GEA

Screen 5. ERAS System File card

The registration subsystem allows managers to register new ENTRY LEVELs as daughters to the ERAS SITE ROOT LEVEL. These have the generic name "Course". When a new ENTRY LEVEL is created, its given name is first verified as being unique, then the manager is prompted to enter the author's name, address, phone number(s), etc. The system then creates an ENTRY LEVEL data structure, with the given name of the new "Course", which is abstracted by the SITE ROOT LEVEL (Figure 9). In turn, the new ENTRY LEVEL abstracts its own LEVEL data structure named "ROOT" and an "Info" data structure that holds the "Course" version number, creation date/time stamp, last modification date/time stamp, and the author's name and registration data. The "ROOT" LEVEL abstracts an empty CONTROL MODULE data structure and an "Info" data structure for its version number and creation and modification date/time stamps.

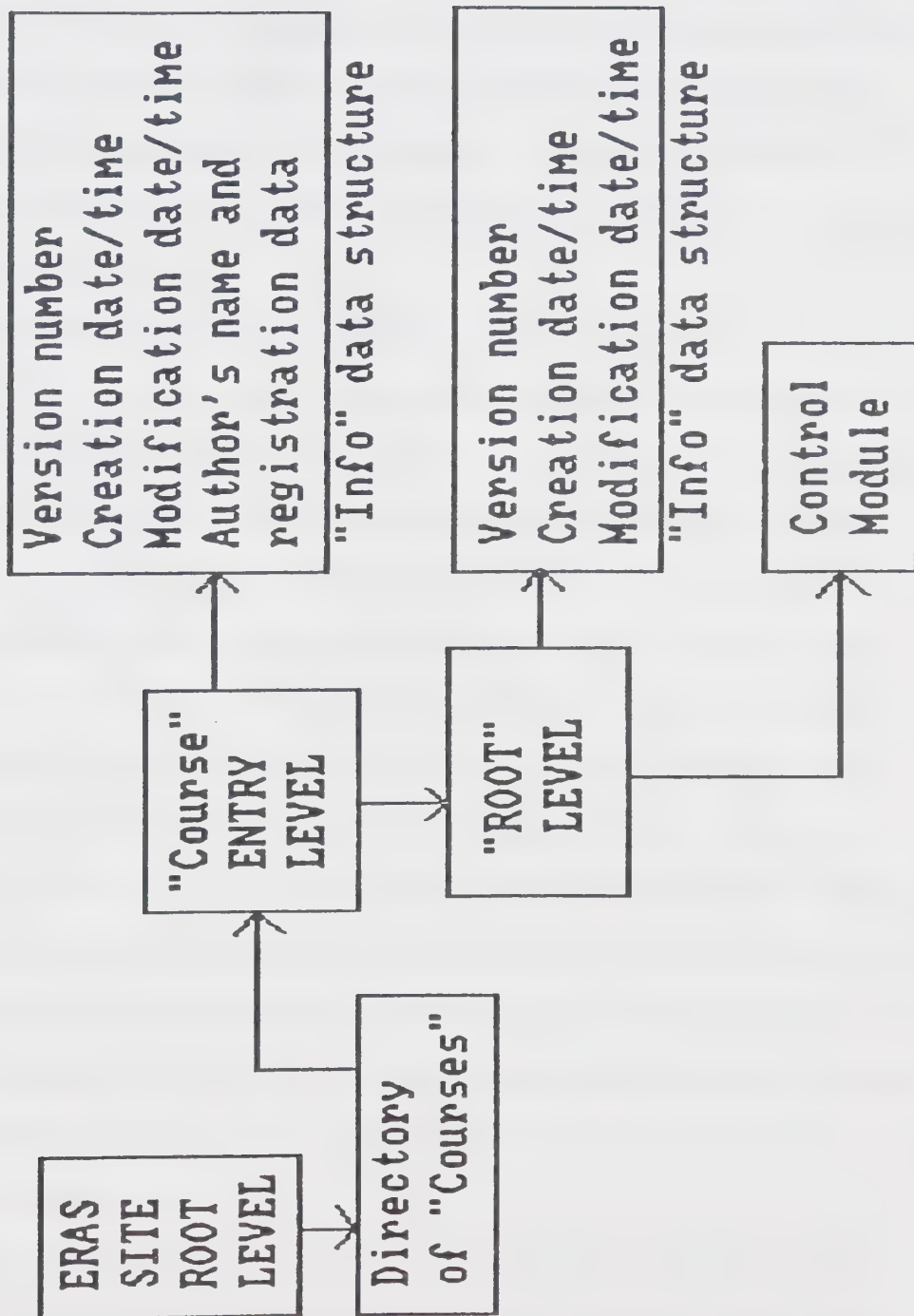
5.9 The Menus and the List Selector

Before describing the file card and tree structure editors, two tools used in the authoring environment will be introduced. These are the pull down menu system and the list selector.

5.9.1 The Pull Down Menus

The same pull down menus are available at all times to the author during the authoring session. When the menu button (the right button) on the mouse is depressed and held down the title bar at the top of the screen changes to a menu bar with a list of menu titles. The menu bar titles are: Project, Exit to, Edit, Create, Directory, and Print. Placing the mouse pointer over one of these titles causes a list of menu items for that menu to drop down. The mouse pointer can be moved down the list causing any item pointed to, to become highlighted. Some menu items, when highlighted, display pop up sub-menus whose items may also be highlighted.

Figure 9. New "Course" Structures



Releasing the menu button on the mouse while an item or sub-item is highlighted causes that item to be selected and the system to take an appropriate action. See Screen 14 for an example of a pull down menu.

Table 3 lists all the pull down menus and sub-menus. In the table, a capital letter in italics following a menu item indicates that that item can be selected directly from the keyboard by holding down a *command* key (the right *AMIGA* key) and pressing that letter key. On the screen this is indicated by displaying an icon for the *command* key followed by the capital letter. The letter has a mnemonic relationship to the menu item. A menu item followed by ">>" indicates that that item has a pop up sub-menu.

Menu items that are inactive are ghosted. All items in the "Project" menu are always active. Selecting the "Run" item invokes the student execution mode at the *LEVEL* being edited. The second item is used to switch between the file card editor and the tree structure editor. While in the file card editor it displays "Tree Editor *T*", and in the tree structure editor it displays "File cards *F*". Selecting "Quit" causes a requester to pop up with the question "Save changes?" and three buttons: "Yes", "No", and "Cancel". Selecting "Yes" or "No" causes the system to take the appropriate action and then exit to the ERAS system file card. Selecting "Cancel" causes the "Quit" operation to be abandoned. Selecting the "About" menu item displays one or two file cards with the author's name and registration data. Selecting "Help" or pressing the *Help* key brings up the help system. (The help system was not part of the prototype implementation.) The "Recover" item allows the author to abandon changes made to the *LEVEL* or module being edited and replace that data object with an earlier saved version. Selecting this item causes a confirmation requester to pop up.

The "Exit to" and "Create" menus will be discussed as part of the file card editor. The "Edit" menu and the first three items of the "Directory" menu will be

Table 3. Pull Down Menus

Project		Exit to		Edit
Run	<i>R</i>	Module	<i>M</i>	Transfer
Tree Editor	<i>T</i>	Directory	<i>D</i>	Clone
Quit	<i>Q</i>	Level	<i>L</i>	Empty
About		Previous	<i>P</i>	
Help		Root	<i>O</i>	
Recover		Entry	<i>E</i>	
		System		

Create	Directory	Print
Documentation	Alphabetical	Module
Generic	Logical	Level >>
Next Level	Edit	Data
Modules >>	ToC >>	All
Content	1	ToC >>
Display	2	1
Input	n...	2
Answer	all	n...
Menu		all
Data >>		
Types		
Variables		
Constants		
Functions		
Support >>		
Routines		
Text files		
Drawings		
Graphs		
Pictures		

discussed as part of the list selector. The "ToC >>" menu item of the "Directory" menu is used to display a "Table of Contents" type list of LEVEL generic names, logical order numbers, and given names starting at the current level. The sub-menu items determine the depth of the listing: "1" displays one level, "2" displays two levels, "n..." interrogates the author for a whole number and displays that number of levels, and "all" displays all levels. The "Print" menu was not part of the prototype implementation but is intended to provide a flexible and formatted listing facility of modules, data definitions, and tables of contents in any future development.

5.9.2 The List Selector

The list selector is invoked by the file card editor to create, list, move, rename, delete, or select identifiers of some specific type. Examples of its use are shown on Screens 6, 9, and 11. It is sometimes accompanied by an auxiliary list to the right as in Screens 9 and 11. Both the list selector and auxiliary list have a descriptive title bar. The current identifier is highlighted in inverse video and is the identifier upon which actions may be taken as outlined below. A different identifier may be selected to be the current identifier by moving the mouse pointer over the desired identifier and pressing the select (left) mouse button. The list of identifiers is terminated by a dummy identifier: "END OF LIST". This "identifier" may be made the current identifier for the "Insert" action described below.

If the list of identifiers is too long to display on one screen then the three "gadgets" on the right side of the list selector can be used to move up and down the list. The two "arrow" gadgets can be selected to move up or down one item at a time. However, holding down the select button causes the list to scroll. The third gadget is called a "proportional" gadget. It displays a rectangular "slider" that graphically shows that proportion of the list that is currently displayed. Clicking above or below the slider moves the list one screen-full up or down. "Dragging" the

slider with the mouse pointer also allows the displayed portion of the list to be changed.

The list selector can operate in three modes: alphabetical, logical, and edit. The mode is designated by selecting one of the first three menu items in the "Directory" menu (Table 3). The current mode is indicated by a small "check mark" in front of the corresponding menu item. The logical mode is the default. In the logical and edit modes the author determines the order of the identifiers in the list. In these modes the author may insert new identifiers anywhere in the list and may move identifiers to new positions in the list. Using the "Edit" menu the author may "Transfer" or "Clone" an identifier (and its data object) to the corresponding list selector in another LEVEL. In the edit mode the author may also rename or delete an identifier. An identifier, and its corresponding data object, may only be deleted if its use count is zero or, for a LEVEL, it is a leaf node. In the alphabetical mode the identifiers are listed in alphabetical order. In all modes an identifier may be selected and passed to the file card editor for some further action.

There are four action buttons at the bottom of the list selector: Select, Move, Insert, and Cancel. In alphabetical mode (as in Screen 6) the "Move" and "Insert" buttons are inactive (ghosted) since these operations are not permitted in this mode. Whenever the "END OF LIST" dummy identifier is made the current identifier, the "Select" and "Move" buttons are inactive since these operations are not possible on this "identifier". In edit mode the "Select" button is renamed "Edit" and the "Cancel" button is renamed "Exit".

Clicking on "Select" closes the list selector and passes the current identifier to the file card editor for further action. Clicking on the "Cancel/Exit" button closes the list selector and abandons the invoking operation of the file card editor.

The action of the "Move" button depends on the settings in the "Edit" menu of Table 3. If both the "Transfer" and "Clone" menu items are *off* (no "check mark" in

front of them) then the move operation is *within* the list selector. If either is *on* then the operation is *between* corresponding list selectors of other LEVELs.

"Transfer" and "Clone" are mutually exclusive boolean switches, that is, they may both not be *on*. They are toggled *on* and *off* by normal menu selection. If one is toggled *on* the other is automatically switched off. The default is both *off*.

The following describes what happens when the "Move" button is clicked on. For the *within* selector move, the current identifier is removed from the list and placed in a temporary buffer. The "Select/Edit" and "Move" buttons are made inactive since only one move operation may proceed at a time. Another current identifier may then be selected on the list. By clicking on the "Insert" button, the moved identifier is inserted in front of the current identifier and becomes the new current identifier. The "Select/Edit" and "Move" buttons are then made active again. If the temporary buffer contains an identifier when "Cancel/Exit" is clicked on, that identifier is inserted in front of the current identifier before the list selector is closed.

For the "Transfer" move, the current identifier is removed from the list and it and its data object are placed in a permanent buffer. For the "Clone" move, the current identifier is not removed from the list but a complete copy of its data object along with its name is placed in the permanent buffer. This buffer is saved between sessions. The "Transfer" and "Clone" menu items are removed from the "Edit" menu and replaced in the menu list by the type name and identifier of the data object in the permanent buffer. This acts as a reminder to the author of what is in the buffer and prevents further *between* selector moves until the data object in the buffer is inserted into another list. The buffer may also be emptied by selecting "Empty Buffer" from the "Edit" menu. Once the buffer is empty the "Transfer" and "Clone" menu items are restored.

A new identifier may only be inserted into a list when the "Move" button is active, that is, when there is nothing in the temporary buffer. When the "Insert"

button is clicked on, a string requester pops up into which the author may type the name of the new identifier. If the identifier is unique to the list it is inserted in front of the current identifier and becomes the new current identifier. If it is not unique the author is informed by a pop up "alert" message. If the list is full the author is also informed by a pop up "alert" message.

If the "Edit" button is clicked on, the list selector is closed and a requester pops up naming the current identifier and displaying three buttons: Select, Rename, and Delete. The action of the "Select" button is the same as for the logical mode. If the "Rename" button is clicked on, a string requester pops up with the identifier displayed. The identifier then may be edited. If a null string is returned then the identifier remains unchanged. Otherwise, the new identifier is checked for uniqueness in the list before allowing the old identifier to be changed. An "alert" message advises the author if the proposed new identifier name is not unique. Two actions are possible if the "Delete" button is clicked on. If the data object may be deleted then a requester pops up to confirm the deletion. If it may not be deleted an "alert" message pops up to give the reason why deletion is denied.

5.10 The File card Editor

After "double clicking" on the ERASAuthor icon in the ERAS system file card, the author enters the file card editor which presents a grey background screen titled "Educational Research Authoring System - Course Filecards". On this screen is displayed a list selector with the title "Select Course" and an alphabetical list of all the ENTRY LEVELs of the SITE ROOT LEVEL (Screen 6).

After selecting one of the "Courses" the list selector is replaced by the course's ENTRY LEVEL file card (Screen 7) with a title displaying "Course:" followed by the course name and version number. On the card is the creation and modification

Select Course

MathDrill

Mathematics30

SpellQuiz

Test1

Test2

***** END OF LIST *****

Select

Move

Insert

Cancel

Screen 6. Course Selection

Created: 26.07.1992 22:41:59

Modified: 26.07.1992 22:41:59

Generic Names

Documentation

Root Level

Screen 7. ENTRY LEVEL File Card

date/time stamps for the ENTRY LEVEL and three buttons: Generic Names, Documentation, and Root Level.

If there are no generic names defined then the "Generic Names" button will be inactive (ghosted) and the "Generic" menu item in the "Create" menu will be active. Otherwise, the reverse will be true.

If no documentation file has been created then the "Documentation" button will be inactive and the "Documentation" menu item in the "Create" menu will be active. Otherwise, again, the reverse will be true. All other items in the "Create" menu will be inactive since they do not apply to the ENTRY LEVEL.

The "Root Level" button is always active.

To create generic names or documentation, the author would select the appropriate item from the "Create" menu. To edit existing generic names or documentation the appropriate button would be click on.

Selecting "Documentation" brings up a system text editor that may be used to create and modify a documentation file of unlimited length. The file is automatically loaded on entry to the text editor and saved on exiting back to the file card editor.

Selecting "Generic Names" causes a requester to pop up (Screen 8). It has the title "Generic Levels - Identifiers" and three buttons: Edit, List, and Print. One of these must be selected. Their actions are each explained below. After the selected action is completed the author is returned to the ENTRY LEVEL file card.

Clicking on the "List" button brings up a file card, with the same title as the requester, that gives a "Table of Contents" type list of all generic names and their aliases. Screen 9 presents an example. This example shows a "Course" that is made up of "Units" and "Exams", with "Units" made up of "Topics" and "Tests", "Topics" made up of "Lessons", "Practices" and "Quizzes", "Lessons" made up of "Sections", "Practices" made up of "Problems", "Quizzes" made up of "Questions", "Tests" made up of "Questions", and "Exams" made up of "Parts" which, in turn, are made up of

Educational Research Au

Course: Mathematics38

Created: 26.07.1992

Modified: 26.07.1992

Generic Names

Documentation

Root Level

Generic Levels - Identifiers

Edit

List

Print

Screen 8. Generic Names Requester

Educational Research Authoring System - Course Filecards

Course: Math Generic Levels - Identifiers

Created:

Modified:

Generic Name

Course

Unit

Topic

Lesson

Section

Practice

Problem

Quiz

Question

Test

Question

Exam

Part

Question

End of List - Press RETURN to continue.

Screen 9. Generic Names List

"Questions". Clicking on the "Print" button prints this list of generic names and their aliases on the system printer.

Clicking on the "Edit" button brings up a list selector and a companion auxiliary list (Screen 10). The title of the auxiliary list is "Generic Levels & Names". During the editing session it displays the current path down the generic names tree structure. The title of the list selector is the name of the current generic tree node followed by the level depth. The list selector is used to create and modify the generic name and its aliases descending from that node. The first name in the logical list is the generic name for the next level. It is followed by zero or more aliases. Selecting a name from the list causes a move down that branch of the generic names tree. Clicking on "Cancel" produces a move up the tree one level to the parent node and eventually to the ENTRY LEVEL file card. Screen 10 shows the display after descending the tree structure shown in Screen 9 to the generic name "Topic" at level 2 of the tree. The level *below* "Topic" has the generic name "Lesson" which has two aliases, "Practice" and "Quiz".

Clicking on the "Root Level" button of the ENTRY LEVEL file card causes a LEVEL file card to replace the ENTRY LEVEL file card (Screen 11). This is the root level for the "Course". The title of the file card is "Course:" followed by the course name. The title of all other LEVEL file cards is the generic name for that level, followed by the level's logical order number, a colon, and the level's given name (see Screen 13).

The top line of each LEVEL file card displays the creation and modification date/time stamps for that level. Below this are two blue background rectangles outlined in yellow. The rectangle on the left displays the level depth number, the level's version number, and buttons to select all non-directory data objects belonging to this level: Control, Defaults, Screen, and Documentation. As in the ENTRY LEVEL file card, the "Documentation" button may be inactive if no documentation file has

Educational Research Authoring System - Course Filecards

Course: Mathen Topic - Level 2

Created: 26
Modified: 26
Generic Names

Lesson
Practice
Quiz
XXXXX END OF DIS XXXXX

Generic Levels & Names

- 0 - Course
- 1 - Unit
- 2 - Topic



Select Move Insert Cancel

Screen 10. Generic Name Creation

Educational Research Authoring System - Course Filecards

Course: Mathenatics88

Created: 26.07.1992 22:42:00 Modified: 26.07.1992 22:42:00

Level 0

Version 1

Control

Defaults

Screen

Documentation

Directories -

Next Level

Modules:

Content

Display

Input

Answer

Menu

Data:

Types

Variables

Constants

Functions

Support:

Routines

Text

Drawings

Graphics

Pictures

Screen 11. LEVEL File Card

been created for this level. The "Control" button leads to the CONTROL MODULE for this level while the "Defaults" and "Screen" buttons lead to settings and switches used by the DISPLAY language. The "Control", "Defaults", and "Screen" buttons are always active.

The rectangle on the right contains buttons to all directories belonging to this level. This rectangle has a button for the "Next Level" directory and three sub-rectangles, outlined in green, titled Modules, Data, and Support. The "Modules" sub-rectangle has buttons for the Content, Display, Input, Answer, and Menu directories. The "Data" sub-rectangle has buttons for the Types, Variables, Constants, and Functions directories. The "Support" sub-rectangle has buttons for the Routines, Text, Drawings, Graphs, and Pictures directories. These directory names also appear in the "Create" menu. The button of an empty directory will be inactive while its corresponding menu item will be active. Likewise, the button of a non-empty directory will be active and its corresponding menu item will be inactive. While in a LEVEL file card, the "Generic" menu item will always be inactive since generic names may only be created while in the ENTRY LEVEL file card.

To create an object in an empty directory the author selects the appropriate item from the "Create" menu. This will open the directory to allow a new object to be created. To edit an existing data object a click on the appropriate directory button would be made. This will bring up a list selector containing the identifiers for all objects in that directory. From this list selector the desired object may be selected.

Screen 12 demonstrates this for the "Next Level" directory. Before this screen the author had descended the course tree structure by selecting "Unit 2: Logarithms" from the course's "Next Level" directory, then "Topic 3: Laws" from Logarithms' "Next Level" directory. When the author selected Laws' "Next Level" directory a list selector and a companion auxiliary list appeared as in Screen 12.

Educational Research Authoring System - Course Filecards

Course: Mathen

Unit 2: Logan

Topic 3: Laws

Created: 27

Level

Version

Control

Default

Screen

Printed

Next Level List

Derive the Laws

Apply the Laws

1 Log Laws

2 Log Laws

***** END OF LIST *****

Generic Path & Local Names

0 - Course

1 - Unit

2 - Topic

0 Lesson

1 Practice

2 Quiz

Select Move Insert Cancel

Screen 12. Next Level Selector

The title of the auxiliary list is "Generic Path & Local Names". Above the double line is the level depth numbers and generic names for all levels for the current level and above. Below the double line is the index number and name for the next level's generic name and its aliases. This shows the generic name to be "Lesson" (index 0), alias 1 to be "Practice" and alias 2 to be "Quiz". (Refer back to Screens 8 and 9.)

The title of the list selector is "Next Level list". A number before an identifier is the index to an alias generic name for that object. A number outside of the range of alias indexes is treated as zero. Since this is a logical list the generic name, logical order number, and given name for each object at the next level would be: Lesson 1: Derive_the_Laws, Lesson 2: Apply_the_Laws, Practice 1: Log_Laws, and Quiz 1: Log_Laws. "2 Log_Laws" is shown as the current identifier. After clicking on the "Select" button a LEVEL file card for "Quiz 1: Log_Laws" would appear cascaded one row down and one column to the right as in Screen 13.

The author may click on any visible LEVEL file card to move back up the tree to that level. The author may also use the "Exit to" menu (Screen 14). The "Module" menu item is only used when editing some aspect of a module. Selecting it causes a return to the module's file card. Selecting the "Directory" menu item causes a return to the most recently used directory list selector. The "Level" menu item is only used when editing an object belonging to a level. Selecting it causes a return to the current LEVEL file card. Selecting the "Previous" menu item causes the current LEVEL file card to be removed and reinstates the previous LEVEL file card. Selecting the "Root" menu item causes all file cards except the root LEVEL file card to be removed. Selecting the "Entry" menu item causes all file cards to be removed and reinstates the ENTRY LEVEL file card. Selecting the "System" menu item causes an exit from the file card editor to the ERAS System file card. This is the same

Educational Research Authoring System - Course Filecards

Course: Mathematics30

Unit 2: Logarithms

Topic 3: Laws

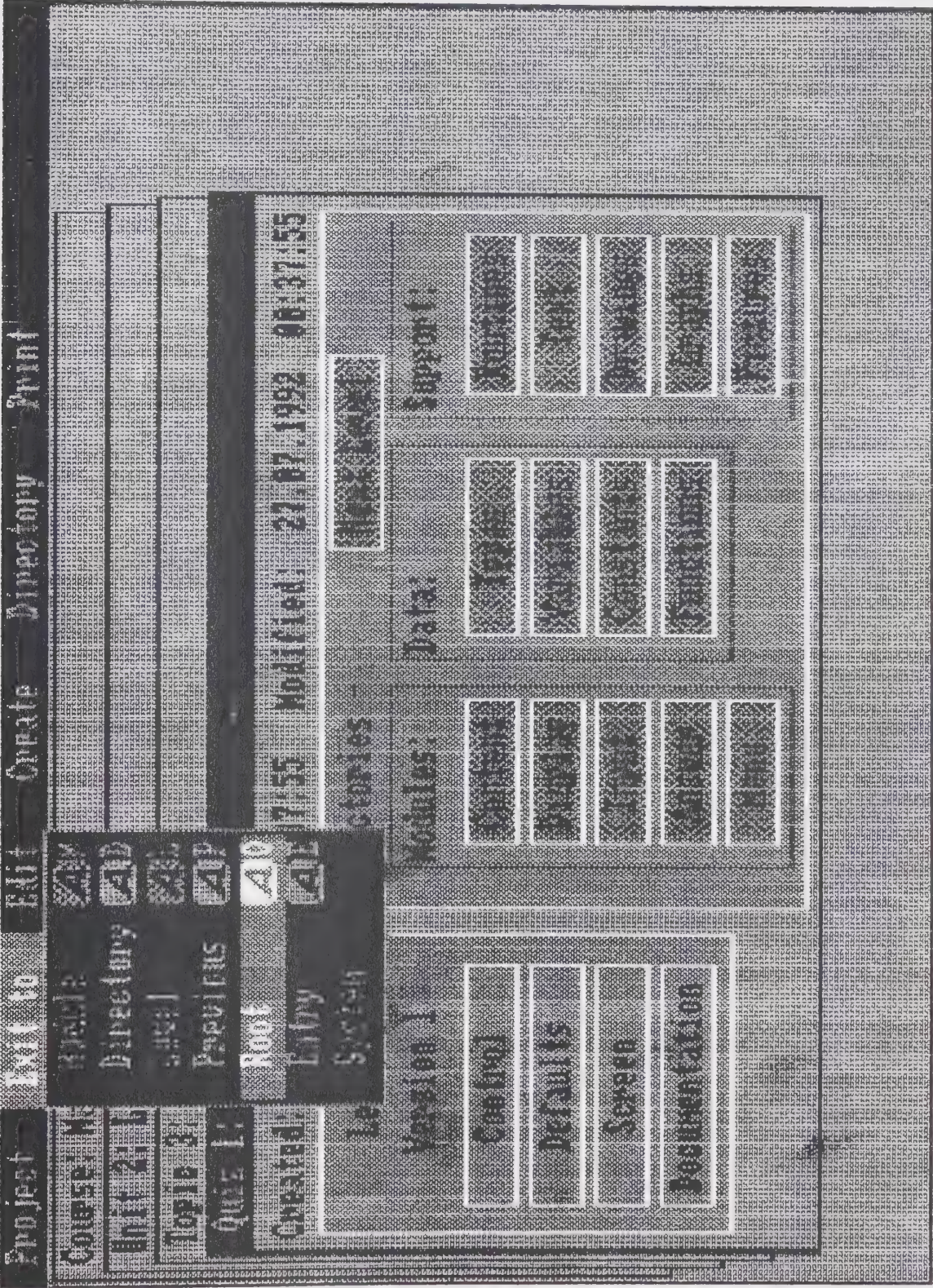
Unit 1: Log Laws

Created: 27.07.1992 00:37:55 Modified: 27.07.1992 00:37:55

Level 3 Version 1 Control Defaults Screen Documentation	Directories - Modules: Content Display Input Answer Menu	Data: Types Variables Constants Functions	Support: Routines Text Drawings Graphics Pictures
--	--	---	--

Screen 13. Cascade of LEVEL File Cards

Screen 14. "Exit to" Menu



action as the "Quit" menu item in the "Project" menu. The "Exit to" menu is context sensitive. Only those menu items that could be selected are active.

5.11 The Tree Structure Editor

If the author was on Screen 14, with the current LEVEL being "Topic 3: Laws", and selected "Tree Editor" from the "Project" menu then Screen 15 would appear.

This is the tree structure editor. It has a tan coloured background and the title "Educational Research Authoring System - Course Structure".

Each LEVEL data object is represented by an "address label" type selector. This selector design allows the greatest number of tree nodes to be displayed at one time on a medium resolution screen with a minimum amount of data displayed for each node. The first line of each label displays the generic name and logical order number for the LEVEL, the middle line displays the given name, and the bottom line shows the version number and last modification date. The label in the centre of the screen is the current LEVEL and is highlighted in yellow.

The labels above and below the current LEVEL are the siblings of the current LEVEL shown in their logical order relationship to it. The label immediately to the left of the current LEVEL is the parent LEVEL to the current LEVEL. The labels above and below the parent LEVEL are the siblings of the parent LEVEL shown in their logical order relationship to it. With a medium resolution screen, at most, four siblings, two above and two below, can be displayed.

To the right of the current LEVEL is a rectangle with a red border. Inside this rectangle are labels for up to five of the daughter LEVELs to the current LEVEL. They may be either in logical order or alphabetical order depending on the settings in the "Directory" menu. The complete list of daughter LEVELs may be moved up and down in the rectangle by the six right-most buttons at the bottom of the screen. The two "arrow" buttons move the list one label at a time. The "PgUp" and "PgDn"

Educational Research Authoring System - Course Structure

Topic 2
Intro_to_Logarithms
Ver. 1 - 26.07.1992

Lesson 2
Apply_the_Laws
Ver. 1 - 26.07.1992

Unit 1
Polynomial_Functions
Ver. 1 - 26.07.1992

Test 1
Exponents_&_Logs
Ver. 1 - 26.07.1992

Practice 1
Log_Laws
Ver. 1 - 26.07.1992

Unit 2
Logarithms
Ver. 1 - 26.07.1992

Topic 3
Laws
Ver. 1 - 26.07.1992

Quiz 1
Log_Laws
Ver. 1 - 26.07.1992

Unit 3
Sequences_&_Series
Ver. 1 - 26.07.1992

Topic 4
Applications
Ver. 1 - 26.07.1992

*** END OF TABLE ***

Exam 1
Mid_Term
Ver. 1 - 26.07.1992

Test 2
Laws_&_Applications
Ver. 1 - 26.07.1992

Enter

Select

Cancel

Delete

Move

Insert

^

✓

PgUp

PgDn

Top

End

Screen 15. Tree Structure Editor

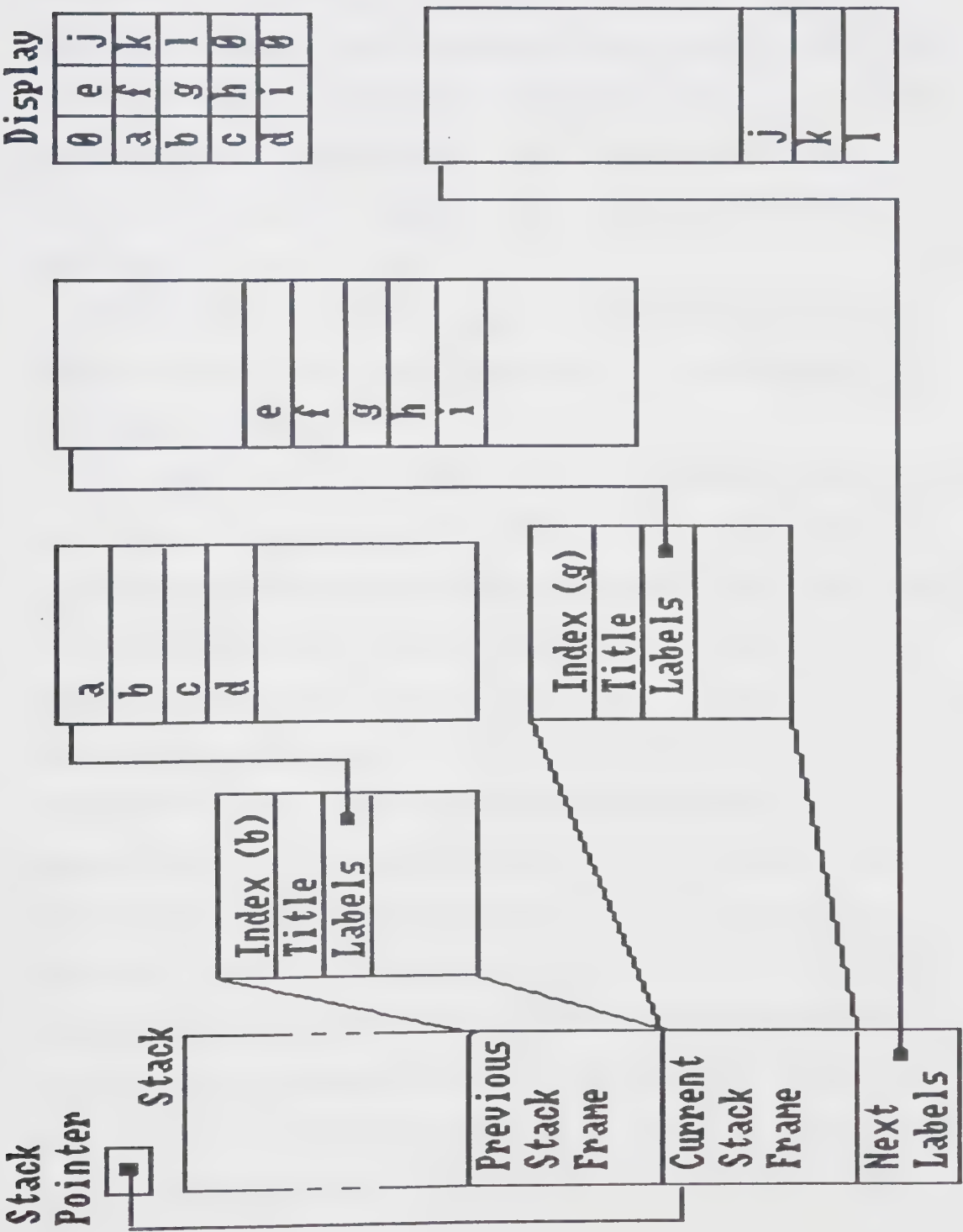
buttons shift the list five labels at a time. The "Top" and "End" buttons move the display to the top and the bottom of the list.

Figure 10 illustrates the internal data structures used to produce the tree structure editor display shown in Screen 15. The "Display" matrix is an array of pointers into the "Labels" lists that contain the actual character data that is displayed in each "label" of the tree structure editor screen display. For example, the "a" in the "Display" matrix points to the "a" in the left most "Labels" list that contains the character strings for the "Unit 1: Polynomial Functions" selector. The "b" contains the strings for "Unit 2: Logarithms" which represents the parent node to the current node which, in turn, is represented by "g" in the second "Labels" list and stands for the strings for "Topic 3: Laws". The "Next Labels" list contains the strings for all the daughter nodes of the current node. The "j" represents the strings for "Lesson 2: Apply the Laws", and so on. A null pointer is represented by a zero in Figure 10. The "Display" matrix is scanned by a screen generation procedure that fetches the strings from the "Labels" lists and writes each "label" selector to its appropriate position on the screen. A null pointer causes any previous "label" in that position to be erased from the screen.

The "Stack Frames" of Figure 10 represent the "LEVEL Stack Frames" of Figure 8. Each "LEVEL Stack Frame" contains, among other data and pointers, the LEVEL's "Title", which is used when producing the cascade of LEVEL file cards in the file card editor. It also contains a pointer to a "Labels" list containing the label display character strings for that LEVEL and all its siblings in the tree database. The "Index" is an offset into the "Labels" list to the display strings for this particular LEVEL.

Any visible label on the tree structure editor display may be clicked on to become the "action" label. The "action" label is highlighted in blue. The six left-most buttons at the bottom of the screen are the "action" buttons and act on the

Figure 10. Tree Display Stack Frames and Label Lists



"action" label. Selecting the "Enter" button causes the LEVEL represented by the "action" label to become the current LEVEL and switches the display to the file card editor with the new current LEVEL in its proper cascaded file card presentation. Selecting the "Select" button also causes the LEVEL represented by the "action" label to become the current LEVEL, but places its label in the centre of the tree structure editor display, and properly adjusts the displayed labels to show the new relationships. Selecting the "Cancel" button removes the highlighting from the new "action" label and highlights the previous "action" label. The "Cancel" button then becomes inactive until another label is clicked on. That is, only the information for one previous selection is kept by the editor.

Whenever the current LEVEL is changed the stack is updated to reflect the change, the "Display" matrix is then updated, and, if in the tree structure editor, the screen display is regenerated.

The "Delete", "Move", and "Insert" buttons work only with the daughter labels in the red rectangle. They work in exactly the same way as the corresponding buttons of the list selector, including their interactions with the "Edit" and "Directory" menus.

If it is inappropriate for a button to be used then it is made inactive and ghosted. For example, if no label has been clicked on to be the "action" label then the six left-most buttons are inactive. If the "action" label is outside the red rectangle then the "Delete", "Move", and "Insert" buttons are inactive. If the daughter labels are at the top of the list then the "up-arrow", "PgUp" and "Top" buttons are inactive. If the daughter labels are at the bottom of the list then the "down-arrow", "PgDn" and "End" buttons are inactive. If a daughter "action" label is not the label for a leaf node then the "Delete" button is inactive. If the daughter list is in alphabetical order then the "Move" and "Insert" buttons are inactive since these actions are only permitted on logical lists where the order of items may be changed and new items inserted.

While in the tree structure editor the "Tree Editor" menu item of the "Project" menu is changed to the "File Cards" menu item. Selecting this item transfers control to the file card editor. On the screen, the cascade of LEVEL file cards is produced by descending the stack from the ROOT LEVEL stack frame through all the LEVEL stack frames and creating a blank LEVEL file card for each LEVEL with just the "Title" of the LEVEL displayed at the top of the file card. When the current LEVEL stack frame is reached, a complete LEVEL file card for the current LEVEL is produced at the end of the cascaded display.

In the "Exit to" menu the "Previous", "Root", "Entry", and "System" menu items may be active. Selecting "Previous" makes the parent LEVEL the current LEVEL, places its label in the centre of the screen, and adjusts all the labels appropriately. Selecting "Root" makes the root LEVEL the current LEVEL, places its label in the centre of the screen, and adjusts all the labels appropriately. Selecting "Entry" causes an exit from the tree structure editor and reinstates the ENTRY LEVEL file card. Selecting "System" causes an exit to the ERAS system file card.

5.12 The Systems Editor

The systems editor is used by the systems programmer for system maintenance and debugging. It has a blue background and the title "ERAS". The ERAS system file card (Screen 5) is the first window of the systems editor. Icons in a window represent data objects that are abstracted from the data object whose window is displayed. Double clicking on an icon causes a window representing that icon to be opened on which are displayed icons for data objects abstracted by that object. The title of a window is the name of the object it represents.

Selecting the "Courses" icon from the ERAS system file card (window) opens a window with file drawer icons for each ENTRY LEVEL (Screen 16). Double clicking on an ENTRY LEVEL icon opens an ENTRY LEVEL window (Screen 17). The "Doc



Spelling



Test2



Test1

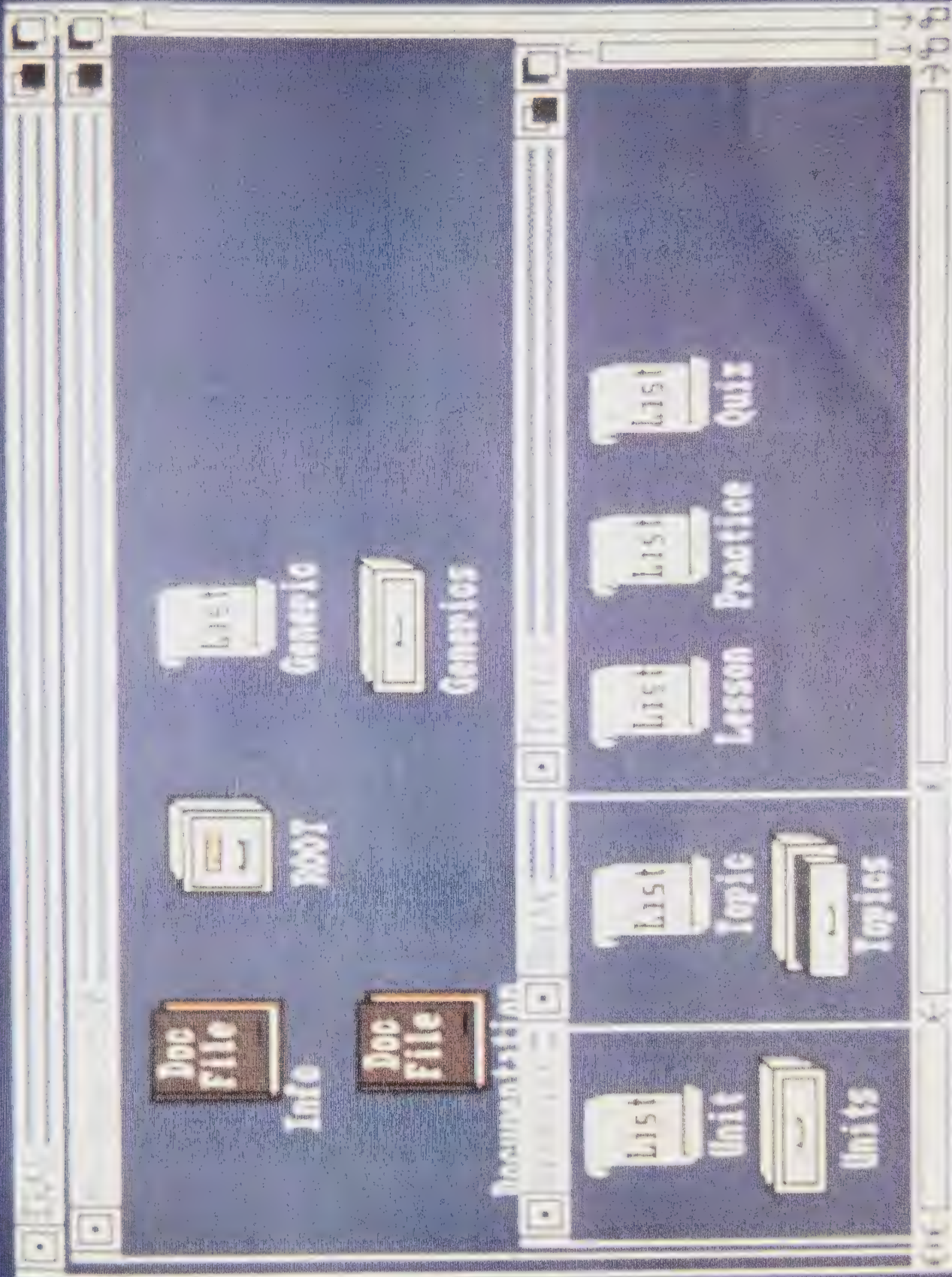


Maths111



Mathematics20

Screen 16. "Courses" Window



Screen 17. ENTRY LEVEL Window

File" icon is used to represent the "Info" record and the "Documentation" file.

Selecting "Info" displays the course version number, creation and modification date/time stamps, and the author registration data. Selecting "Documentation" allows for the editing of the documentation file with the system text editor. The "ROOT" file drawer icon represents the course root LEVEL.

The "List" scroll icons contain lists of identifiers used by the list selector. The small drawer icons hold objects named in the related "Lists". The "Generic" icons represent the first level of the generic names tree structure. Screen 17 shows the windows opened after selecting "Generics", "Units", and "Topics" in that order. Compare this to the list in Screen 9 that shows a "Course" made up of "Units", "Units" made up of "Topics", and "Topics" made up of "Lessons", "Practices", and "Quizzes". Selecting any "List" icon displays the identifiers in the list in their logical order.

Selecting "ROOT" opens a LEVEL window (Screen 18). A "module" scroll icon is used to represent a module data object. All LEVELs have a CONTROL module. Every active directory would have a "List" icon for the identifiers in the directory. The "NextLevel" list is shown. File drawer icons represent each LEVEL data object abstracted from the displayed LEVEL window.

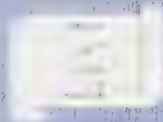
As in any desk top metaphor the systems programmer has considerable freedom in resizing widows, positioning windows, closing windows, and moving icons. This freedom must be supported by a firm knowledge of the relationships of all data objects that make up the ERAS system.



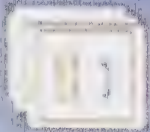
Info



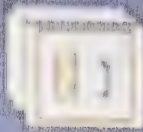
Control



HexView



Sequences_L_Series



Statistics



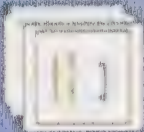
Presentation



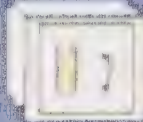
Polynomial_Functions



MidView



Final



quadratic_Relations



Polynomkey

Screen 18. LEVEL Window

6. Conclusions and Assessment

6.1 Expectations Attained

The expectations for this thesis research are specified in section 2.2. These were summarised in that section as follows:

Given a predefined CAI authoring language, the task is to define and implement the following: (1) a hierarchical tree structured database to hold the courseware modules and data specified for the given language, and (2) a visual authoring environment to support the creation and modification of the objects contained in this database. For this research the authoring environment will not include the creation and modification of the language statements or the data items other than what is needed to demonstrate the viability of the system.

The ERAS Authoring Language of the Educational Research Authoring System was chosen as the *predefined* authoring language for this research since it was designed to have its language modules and supporting data definitions reside in a hierarchical database. Interpreters for the six ERAS sub-languages had been implemented in the Elf (Educational Language Facility) system development environment which was under development at the Division of Educational Research Services, Faculty of Education, University of Alberta.

The curtailment of the development of the Elf System because of financial restraints in the Faculty of Education, University of Alberta, severely restricted the completion of this research. Since the predefined CAI authoring language was implemented on the Elf system, it was originally expected that this research project would be completed on the Elf system. A critical component of Elf, the graphic user interface, was only partially implemented for the Digital Equipment (DEC) Gigi colour graphic terminal when the Elf project had to be moved from the decommissioned DEC VAX 11/780 mini-computer of the Division of Educational Research Services to the

Apple Macintosh II microcomputer. This new implementation of Elf was still under development and not available to complete this thesis research.

The courseware storage mechanism, the run-time system and part of the authoring environment were completed in Elf on the DEC VAX before use of this system was withdrawn. The final demonstration of the visual authoring environment was completed on a Commodore *AMIGA* microcomputer using GFABasic with Extend and the Intuition graphic user interface of the *AMIGA* operating system. Prototypes of the user interface for the file card and tree structure editors were completed on the *AMIGA*. A desk top metaphor viewer of the courseware storage system was also developed on the *AMIGA*.

Chapter 4 outlined the design of the research project for a multi-user large scale system. Chapter 5 described the implementations of two single-user prototypes, one in Elf on the DEC VAX and one on the *AMIGA*.

The expectations for the hierarchical courseware database were specified in section 2.2.2. The database was designed by using Elf *media* files and Elf data structures. These were linked together to form the tree structure with Elf *abstractions*. This form was fully implemented in the Elf prototype and emulated in the *AMIGA* prototype by making use of the hierarchical structure of the *AMIGA*'s file directory system. Each "Course" was created with an entry node and a root node. An entry node was termed an ENTRY LEVEL.

The database was a multi-branch tree structure. Each node in the tree was termed a LEVEL. Only leaf nodes could be deleted and any node except a root node could be moved to become the daughter of any other node. An author could traverse the branches of the tree in either direction during an authoring session.

At the ENTRY LEVEL an author could define and modify a set of generic names for each level of the tree below the root LEVEL attached to that ENTRY LEVEL. Each generic name could have any number of aliases. The author supplied each node

in the tree with its own given name. The branches from a node had a logical order determined by the author. New branches could be inserted at any place in this order and the order of existing branches could be changed. Each branch from a node could be referred to by its generic name and logical order number or by its given name.

Each LEVEL had its own Control Module (of the ERAS language) and could have a "Next Level" directory of branches from that LEVEL. A LEVEL could have any or all of the typed directories listed in Table 2 of section 2.2.2. These directories were created and deleted automatically when the first object for a directory was created and when the last object was deleted. All of the directories of Table 2 were provided for in both prototypes but were implemented only in the Elf prototype. Their internal structure is similar to that of the "Next Level" directory which was fully developed in both prototypes.

The scope and visibility of all identifiers was determined by the location of the object it named in the tree. This was enforced by the stack structure of the run-time system. A control module for a LEVEL could only invoke a control module of a LEVEL that was a branch from its own LEVEL. LEVEL stack frames were pushed onto the stack and linked by abstraction while descending the tree and popped from the stack when ascending the tree. Each LEVEL stack frame abstracted all directories belonging to that LEVEL. This permitted identifiers to be searched for first at the local LEVEL and then at each LEVEL up the stack.

Internal documentation text could be attached to each LEVEL data object and could be edited whenever the author was at that LEVEL during an authoring session.

The expectations for the visual authoring environment were specified in section 2.2.3. Two visual authoring modes were created, a tree structure editor and a file card editor.

The tree structure editor was only implemented in the *AMIGA* prototype. It was a window that could be moved over any part of the course tree structure. Each node

in the tree was represented by an "address label" icon that displayed the node's generic name, logical order number, given name, version number, and last modification date.

The hierarchy of the tree was represented by three columns of node labels with up to five labels visible in each column. The label in the centre of the screen represented the current node. Its immediate logical sister nodes were represented by labels above and below it. The label to the left represented its parent node with the parent's sister node labels displayed above and below it in the first column. The last column was used to display labels for the current node's daughter nodes. This column could be scrolled to display all the daughter nodes.

Any visible node could be selected to be the current node or to be entered for modification. A drop down menu selection could be used to make either the current parent node or the root node the current node. Any daughter node that was also a leaf node could be deleted. A daughter node, along with its sub-tree, could be placed into a buffer and inserted elsewhere in the tree. New nodes could be created and inserted anywhere in the set of daughter nodes. New nodes required the specification of its generic name, which could be defaulted, and its given name.

The file card editor was implemented in both the Elf prototype on the DEC VAX and in the *AMIGA* prototype. A file card was used to represent a LEVEL data object. These were displayed in a cascade of file cards that illustrated the relative position of the LEVEL in the tree structured database. Each LEVEL file card displayed the data and selection buttons specified in the expectations of section 2.2.3.

When a directory button was selected a file card for that directory was displayed. The identifiers in the directory could be listed in either logical or alphabetical order. This was determined by a switch set in a pull down menu. An item could be selected for modification, deletion, logical order change, or movement to another directory of the same type in another node. A new item could be inserted

anywhere in the logical list. All directory types were supported in the Elf prototype but only the "Next Level" directory type was implemented in the *AMIGA* prototype. The management system was the same for all directory types but the *AMIGA* prototype did not have access to the ERAS interpreters so there was no way for the directories to be tested other than the "Next Level" directory which was directly part of the creation of the tree structure database.

Authoring functions on the Elf prototype were implemented by cursor movement, controlled by arrow cursor control keys, and a function selection keypad. A diagram of the keypad could be displayed showing the layout and names of the function keys. Authoring functions on the *AMIGA* prototype were implemented by use of a mouse controlled screen pointer, screen selection buttons, and pull down menus. Menu items and selection buttons that were not currently active were *ghosted* and would not become highlighted when pointed to.

The author could point to and select any visible file card from the cascade of file cards to make it the currently displayed and active file card. Using pull down menus the author could also move to the current directory file card, the previous level file card, the root level file card, the entry level file card, the tree structure editor, or the system.

Since the ERAS sub-language interpreters were only available in Elf on the DEC VAX, only the Elf prototype had a run-time environment. The run-time stack was maintained throughout the authoring session. The author could move freely between editing and executing the course. Execution could be interrupted and the author could move up and down the run-time stack and select to edit any LEVEL data object abstracted by the stack frames. Contents of variables abstracted by stack frames could also be examined.

6.2 Conclusions

One might define, for the computer-assisted instruction (CAI) system designer, what might be called an *historical imperative*:

To provide the subject matter expert and instructional designer the most effective and efficient CAI authoring tools possible within the current limits of hardware availability and software development theory and practice.

It has been the pursuit of this goal that has motivated this research.

A review of the historical development of CAI from the *creative breakthrough* in 1958, through the *replication* period [1959-66], the *empirical* period [1967-74], the *theoretical* period [1975-82], and the *automation* period [1983-90] was carried out. The evolution of the sequence control and courseware management aspects of CAI languages and authoring systems was examined. These two aspects are the focus of this thesis. It was found that almost all CAI languages and authoring systems tend to provide a two level system of management and control: a within-file system and a separate and distinct between-file system. It was concluded that a more unified multi-level system of management and control features in a CAI system would enhance courseware organization, design and development. Ideas for the design of a large scale CAI system were also contributed from the following areas of computer science: the concept of abstraction, visual programming, human-computer interaction, and graphical user interfaces.

The design of a large scale, multi-user CAI system was proposed based on a modular CAI language, ERAS, which has six sub-languages: CONTROL, CONTENT, DISPLAY, INPUT, ANSWER, and MENU. The system supported a hierarchical courseware data base and a visual authoring environment. It was designed to have a unified look and feel for all classes of users, to incorporate features that support user and courseware registration, and to assist authors at the design stage of courseware development.

Two single user prototypes were developed to test some of the design features and the user interface.

6.3 Assessment

An obvious advantage of the approach used in this research is that the system allows the author to visualize the relationships among the various courseware components. This is especially useful during the design stage of courseware development.

One of the necessary tools that needed to be developed for this research project was an identifier list editor and selector. The original editor made use of the Elf list data structure to maintain various lists of identifiers and to abstract the data objects associated with each identifier. The *AMIGA* version made use of the Intuition windows, boolean gadgets, and proportional gadget to maintain the list display. The identifier lists were stored in logical order in sequential files on disk and brought into main memory when required. GFABasic has a built in *quicksort* algorithm that was used to sort lists into alphabetical order when required. This list editor and selector is described in section 5.9.2.

The Elf *abstraction* mechanism for dynamically linking data objects and the Elf *media* structure for dynamically creating data objects in relocatable blocks of storable memory proved to be invaluable tools for the implementation of the thesis research design. These tools were successfully emulated on the *AMIGA* by using the hierarchical directory structure of the *AMIGA*'s operating system. This meant that the courseware database was dynamically created and modified on secondary storage leaving primary memory free to develop modules and to maintain a system of pointers into the tree structure. During authoring, data on all tree nodes that could be reached by moving up the tree from the current node were maintained on a stack in

primary memory. This allowed the tree structure editor to rapidly display the node data as the author moved about the tree database.

The techniques used on the *AMIGA* could be used to port this CAI system to any platform that supports a hierarchical directory structure on secondary storage and has a graphic user interface.

6.4 Future Research

One of the weaknesses of the design given in this thesis is the computation aspect. The stated objective was to provide a "powerful computation capability" (see section 4.5). Although useful for many types of applications, having only one numeric type (three decimal place fixed point real) and only two type constructors, RECORD (with no pointer type) and LIST, could limit an author's flexibility in designing some types of courseware.

The STRING type provided is based on the very powerful Elf STRING type and is more than adequate for the CAI environment.

Ultimately, the author should have access to facilities similar to those offered by the more advanced languages like Modula-2. Modula-2 offers the scalar types of INTEGER, CARDINAL, REAL, BOOLEAN, and CHARACTER as well as enumeration types and subrange types. It also provides type constructors for ARRAYs, RECORDs, SETs, and POINTERs.

Section 3.1 suggested the usefulness of abstract data types in a CAI environment. Abstract data types are also necessary for object oriented programming. They could be implemented in the design proposed in this thesis by a new construction called a PACKAGE. (Since *module* is used in this design in a way that differs from its use in Modula-2, the name *package* has been borrowed from Ada.) Each instance of a LEVEL could contain a directory of PACKAGEs. A PACKAGE would EXPORT

an *opaque* type and a set of operations on that type. The implementation of the type and its operations would be *hidden* from those IMPORTing them. PACKAGES would obey the scoping rules of the hierarchical system. As in Modula-2, objects could be EXPORTed from a PACKAGE as a qualified or an unqualified identifier. That is, an object EXPORTed as a qualified identifier would need its name prefixed by the PACKAGE name.

As soon as the new version of Elf and its graphic user interface are completed on the Macintosh II, the software developed as part of this thesis research should be ported from the VAX and AMIGA implementations to the Macintosh II and integrated. When Elf provides network facilities, a multi-user version of this research project (with an integrated registration system) should be implemented.

The validity of the system for the development and delivery of courseware needs to be ascertained. Because of the stated goals of the design, this would have to be a long term project. Authors, both experienced and inexperienced, would have to develop courseware on the system and have it tested by students. The observations of all users would be used in an iterative process of step-by-step evaluation and improvement leading to the use of the system in large scale multi-author CAI projects.

BIBLIOGRAPHY

- Alessi, S.M., & Trollip, S.R. (1985). *Computer-based instruction: Methods and development*. Englewood Cliffs, NJ: Prentice-Hall.
- Allen, M., & Szabo, M. (1990, June). *Dimensions of authoring aids intelligence*. Paper presented at the International Conference on Computer Assisted Learning, Hagen, Germany.
- Atherton, R. (1982). *Structured programming with COMAL*. Chichester, England: Ellis Horwood.
- Authorware. (1987). *Course of Action: Reference manual*. Minneapolis, MN: Author.
- Authorware. (1989). *Authorware Professional*. Minneapolis, MN: Author.
- Avner, A., Smith, S., & Tenczar, P. (1984). CBI authoring tools: Effects on productivity and quality. *Journal of Computer-Based Instruction*, 11, 85-89.
- Avner, R.A., & Tenczar, P. (1969). *The TUTOR manual*. Urbana, IL: University of Illinois, Computer-based Education Research Laboratory.
- Azuma, M., Tabata, T., Oki, Y., & Kamiya, S. (1985). SPD: A humanized documentation technology. *IEEE Transactions on Software Engineering*, 11, 945-953.
- Bagnall, K.M., Szabo, M., Halls, S., & Jensen, W. (1984). The evolution of an authoring system on PLATO. *Journal of Computer-Based Instruction*, 11, 76-77.
- Barker, H.A., Chen, M., Jobling, C.P., & Townsend, P. (1987). Interactive graphics for the computer-aided design of dynamic systems. *IEEE Control Systems*, 7(3), 19-25.
- Barker, P. (1987). *Author languages for CAL*. London: Macmillan Education.
- Basili, V.R., & Baker, F.T. (1981). *Tutorial on structured programming: Integrated practices*. Los Alamitos, CA:IEEE Computer Society Press.
- Beaumont, J.G. (1985). Speed of response using keyboard and screen-based microcomputer response media. *International Journal of Man-Machine Studies*, 23, 61-70.
- Bell & Howell. (1981). *PASS: Professional Authoring Software System*. Weston, ON: Author.
- Beretta, M., Mussio, P., & Protti, M. (1986). Icons: Interpretation and use. 1986 *IEEE Computer Society Workshop on Visual Languages* (pp. 149-158). Washington: IEEE Computer Society Press.
- Berzins, V., Gray, M., & Naumann, D. (1986). Abstraction-based software development. *Communications of the ACM*, 29, 402-415.

- Bitzer, D.L., Hicks, B.L., Johnson, R.L., & Lyman, E.R. (1967). The PLATO system: Current research and developments. *IEEE Transactions on Human Factors in Electronics*, 8, 64-70.
- Bitzer, D.L., Braunfeld, P.G., & Lichtenberger, W.W. (1962). PLATO II: A multiple-student, computer-controlled, automatic teaching device. In J.E. Coulson (Ed.), *Programmed learning and computer-based instruction: Proceedings of the conference on application of digital computers to automated instruction* (pp. 205-216). New York: John Wiley and Sons.
- Boehm, B.W. (1987). Improving software productivity. *IEEE Computer*, 20(9), 43-57.
- Booch, G. (1987). *Software engineering with Ada*. Menlo Park, CA: Benjamin/Cummings.
- Bork, A. (1984). Production systems for computer-based learning. In D.F. Walker, & R.D. Hess (Ed.), *Instructional software: Principles and perspectives for design and use* (pp. 96-114). Belmont, CA: Wadsworth.
- Bournique, R., & Treu, S. (1985). Specification and generation of variable, personalized graphical interfaces. *International Journal of Man-Machine Studies*, 22, 663-684.
- Brahan, J.W., Farley, B., & Orchard, R.A. (1985). *A set of tools for courseware development*. Unpublished manuscript, National Research Council Canada, Division of Electrical Engineering, Ottawa.
- Brahan, J., & Godfrey, D. (1984). *Computer-aided learning using the NATAL language*. Victoria, BC: Press Porcépic.
- Brief history of computer-assisted instruction at the institute for mathematical studies in the social sciences*. (1968). Palo Alto, CA: Stanford University.
- Brown, C.M. (1988). *Human-computer interface design guidelines*. Norwood, NJ: Ablex.
- Brown, G.P., Carling, R.T., Herot, C.F., Kramlich, D.A., & Souza, P. (1985). Program visualization: Graphical support for software development. *IEEE Computer*, 18(8), 27-35.
- Brown, M., Blachier, A., & Yankelovich, G. (1987, July 23). Prototyping information systems with 4GLs. *Computing Canada*, pp. 26-27.
- Brown, M.H. (1988). Exploring algorithms using Balsa-II. *IEEE Computer*, 21(5), 14-36.
- Brown, P.J. (1979). *Writing interactive compilers and interpreters*. Chichester, England: John Wiley & Sons.
- Burns, A., & Robinson, J. (1986). ADDS - A dialogue development system for the Ada programming language. *International Journal of Man-Machine Studies*, 24, 153-170.
- Carey, T. (1982). User differences in interface design. *IEEE Computer*, 15(11), 14-20.

- Chambers, J.A., & Sprecher, J.W. (1984). Computer-assisted instruction: Current trends and critical issues. In D.F. Walker, & R.D. Hess (Ed.), *Instructional software: Principles and perspectives for design and use* (pp. 6-19). Belmont, CA: Wadsworth.
- Chambers, J.A., & Sprecher, J.W. (1983). *Computer-assisted instruction: Its use in the classroom*. Englewood Cliffs, NJ: Prentice-Hall.
- Chang, S.K. (1987). Visual languages: A tutorial and survey. *IEEE Software*, 4(1), 29-39.
- Chaudhary, B.D., & Sahasrabuddhe, H.V. (1985). A study in dimensions of psychological complexity of programs. *International Journal of Man-Machine Studies*, 23, 113-133.
- Collis, B., & Gore, M. (1985). *Combining software engineering and instructional design principles in a new type of course for educators*. Victoria, BC: University of Victoria, Software Engineering/Education Cooperative Project.
- Collis, B., & Muir, W. (1984). *Computers in education: An overview*. Victoria, BC: University of Victoria, Software Engineering/Education Cooperative Project.
- Control Data. (1987). *PCD3 authoring system user manual*. Minneapolis, MN: Author.
- Control Data. (1978). *PLATO author language reference manual*. St. Paul, MN: Author.
- Coulson, J.E. (Ed.). (1961). *Programmed learning and computer-based instruction: Proceedings of the conference on application of digital computers to automated instruction*. New York: John Wiley and Sons.
- Covington, M.A. (1991). *Study keys to computer science*. Hauppauge, NY: Barron's Educational Series.
- Draper, S.W., & Norman, D.A. (1985). Software engineering for user interfaces. *IEEE Transactions on Software Engineering*, 11, 252-258.
- Ellis, H.C., & Hunt, R.R. (1983). *Fundamentals of human memory and cognition* (3rd ed.). Dubuque, IA: Wm. C. Brown.
- Engels, G., Gall, R., Nagl, M., & Schafer, W. (1983). Software specification using graph grammars. *Computing*, 31, 317-346.
- Engels, G., Görgens, M., & Ostrowski, F. (1988). *GFA BASIC interpreter manual* (J. Little, Trans.). San Francisco: Antic Software.
- England, E. (1984). Colour and layout considerations in CAL materials. *Computer Education*, 8, 317-321.
- Ewing, J., Mehrabanzad, S., Sheck, S., Ostroff D., & Shneiderman, B. (1986). An experimental comparison of a mouse and arrow-jump keys for an interactive encyclopedia. *International Journal of Man-Machine Studies*, 24, 29-45.

- Fairweather, P.G., & O'Neal, A.F. (1984). The impact of advanced authoring systems on CAI activity. *Journal of Computer-Based Instruction*, 11, 90-94.
- Feingold, S.L., & Frye, C.H. (1966). *Users Guide to PLANIT*. Santa Monica, CA: System Development Corporation.
- Fisher, F.D. (1982). Computer-assisted education: What's not happening? *Proceedings of the Conference of the Association for the Development of Computer-Based Instructional Systems* (pp. 16-21). Bellingham, WA: Western Washington University.
- Ford, G.A., & Wiener, R.S. (1985). *Modula-2: A software development approach*. New York: John Wiley & Sons.
- Gaines, B.R. (1984). The technology of interaction - dialogue programming rules. In D.F. Walker, & R.D. Hess (Ed.), *Instructional software: Principles and perspectives for design and use* (pp. 115-129). Belmont, CA: Wadsworth.
- Gaines, B.R., & Shaw, M.L.G. (1986a). Foundations of dialog engineering: The development of human-computer interaction (Part 2). *International Journal of Man-Machine Studies*, 24, 101-123.
- Gaines, B.R., & Shaw, M.L.G. (1986b). From timesharing to sixth generation: The development of human-computer interaction (Part 1). *International Journal of Man-Machine Studies*, 24, 1-27.
- Galitz, W.O. (1985). *Handbook of screen format design* (rev. ed.). Wellesley Hills, MA: QED Information Sciences.
- Galotti, K.M., & Ganong, W.F., III. (1985). What non-programmers know about programming: Natural language procedure specification. *International Journal of Man-Machine Studies*, 22, 1-10.
- Gannon, J.D., Hamlet, R.G., & Mills, H.D. (1987). Theory of modules. *IEEE Transactions on Software Engineering*, 13, 820-829.
- Gannon, J.D., Katz, E.E., & Basili, V.R. (1986). Metrics for Ada packages: An initial study. *Communications of the ACM*, 29, 616-623.
- Garraway, R.W.T. (1983). *Microcomputer based computer-assisted learning system: CASTLE*. Unpublished masters dissertation, University of Alberta, Edmonton.
- Gentile, J.R. (1965). *The first generation of computer-assisted instructional systems: An evaluative review*. University Park, PA: Pennsylvania State University, College of Education, Computer Assisted Instruction Laboratory.
- Gillingham, M.G. (1988). Text in computer-based instruction: What the research says. *Journal of Computer-Based Instruction*, 15, 1-6.

- Gillingham, M., Murphy, P., Cresci, K., Klevenow, S., Sims-Tucker, B., Slade, D., & Wizer, D. (1986). An evaluation of computer courseware authoring tools and a corresponding assessment instrument for use by instructors. *Educational Technology*, 26(9), 7-17.
- Glenn, A.D., & Carrier, C.A. (Eds.). (1989). Teacher technology training. *Educational Technology*, 29(3).
- Glinert, E.P. (1986). Towards "second generation" interactive, graphical programming environments. *1986 IEEE Computer Society Workshop on Visual Languages* (pp. 61-70). Washington: IEEE Computer Society Press.
- Godfrey, D., & Sterling, S. (1882). *The elements of CAL*. Victoria, BC: Press Porcépic.
- Goldberg, A. (1984). *Smalltalk-80: The interactive programming environment*. Reading, MA: Addison-Wesley.
- Good, I.J. (1980). Pioneering work on computers at Bletchley. In N. Metropolis, J. Howlett, & G. Rota (Eds.), *A history of computing in the twentieth century: A collection of essays* (pp. 31-45). New York: Academic Press.
- Gore, M., & Collis, B. (1985). *Applying software engineering principles to the development of educational software*. Victoria, BC: University of Victoria, Software Engineering/Education Cooperative Project.
- Grafton, R.B., & Ichikawa, T. (1985). Visual programming. *IEEE Computer*, 18(8), 6-9.
- Hammond, N., & Barnard, P. (1984). Dialogue design: Characteristics of user knowledge. In A. Monk (Ed.), *Fundamentals of human-computer interaction* (pp. 127-164). London: Academic Press.
- Hayes, F., & Baran, N. (1989). A guide to GUIs. *Byte*, 14(7), 250-257.
- Hickey, A.E. (Ed.). (1968). *Computer-assisted instruction: A survey of the literature* (3rd Ed.). Newburyport, MA: Entelek.
- Highway, B. (1989). *Extend: The BASIC extension (V1.3)*. Buffalo, NY: Sunsmile Software.
- Hillelsohn, M.J. (1984). Benchmarking authoring systems. *Journal of Computer-Based Instruction*, 11, 95-97.
- History of the TICCIT system. (1978). *ADCIS News*, 10(5), 32-36.
- Hollnagel, E. (1983). What we do not know about man-machine systems. *International Journal of Man-Machine Studies*, 18, 135-143.
- Honeywell Information Systems. (1981a). *NATAL II language specification*. Willowdale, ON: Author.
- Honeywell Information Systems. (1981b). *NATAL II utilities guide*. Willowdale, ON: Author.

- Hulme, C. (1984). Reading: Extracting information from printed and electronically presented text. In A. Monk (Ed.), *Fundamentals of human-computer interaction* (pp. 35-47). London: Academic Press.
- Hunka, S. (1986). CAL authoring system. *Proceedings of the Fifth Canadian Symposium on Instructional Technology* (pp. 316-319). Ottawa: National Research Council Canada.
- Hunka, S. (1988a). *Computer assisted instruction authoring system* (RIR-88-2). Edmonton, AB: University of Alberta, Division of Educational Research Services.
- Hunka, S. (1988b). *Fifteen years of teaching elementary applied statistics using CAI* (RIR-88-6). Edmonton, AB: University of Alberta, Division of Educational Research Services.
- Hunka, S. (1989). *Design Guidelines for CAI authoring systems* (RIR-89-2). Edmonton, AB: University of Alberta, Division of Educational Research Services.
- Huntington, J.F. (1986). Computer comments. *Educational Technology*, 26(5), 32-33.
- International Business Machines. (1968). *IBM 1500 Coursewriter II, Author's guide*. San Jose, CA: Author.
- International Business Machines. (1977). *Interactive instruction system: General information manual*. (1977). White Plains, NY: Author.
- Isoda, S., Shimomura, T., & Ono, Y. (1987). VIPS: A visual debugger. *IEEE Software*, 4(3), 8-19.
- Jacky, J.P., & Kalet, I.J. (1987). An object-oriented programming discipline for standard Pascal. *Communications of the ACM*, 30, 772-776.
- Jacob, R.J.K. (1985). A state transition diagram language for visual programming. *IEEE Computer*, 18(8), 51-59.
- Jacob, R.J.K. (1986). A specification language for direct-manipulation user interfaces. *ACM Transactions on Graphics*, 5, 283-317.
- Jagodzinski, A.P. (1983). A theoretical basis for the representation of on-line computer systems to naive users. *International Journal of Man-Machine Studies*, 18, 215-252.
- Kamel, R.F. (1987). Effect of modularity on system evolution. *IEEE Software*, 4(1), 48-54.
- Kauffman, D., & Lamkin, C. (1984). Designing schools for tomorrow's technology. *AEDS Monitor*, 23(1,2), 10-15.
- Kearsley, G. (1982). Authoring systems in computer based education. *Communications of the ACM*, 25, 429-437.

- Kearsley, G. (1983). *Computer-based training: A guide to selection and implementation*. Reading, MA: Addison-Wesley.
- Kearsley, G. (1984). Authoring tools: An introduction. *Journal of Computer-Based Instruction*, 11, 67.
- Kearsley, G., Hunter, B., & Seidel, R.J. (1983a). Two decades of computer based instruction projects: What have we learned? [Part 1]. *Technological Horizons in Education Journal*, 10(3), 90-94.
- Kearsley, G., Hunter, B., & Seidel, R.J. (1983b). Two decades of computer based instruction projects: What have we learned? [Part 2]. *Technological Horizons in Education Journal*, 10(4), 88-96.
- Kerr, S.T. (1985). Videotex and education: Current developments in screen design, data structure, and access control. *Machine-Mediated Learning*, 1, 217-254.
- Klass, R. (1984). The TenCORE language and authoring system for the IBM personal computer. *Journal of Computer-Based Instruction*, 11, 70-71.
- Korfhage, R.R. (1986). Browser - A concept for visual navigation of a database. 1986 *IEEE Computer Society Workshop on Visual Languages* (pp. 143-148). Washington: IEEE Computer Society Press.
- Kozma, B.K. (1986). Present and future computer courseware authoring systems. *Educational Technology*, 26(6), 39-41.
- Larkin, J.H., & Simon, H.A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11, 65-99.
- Lemmons, P. (1981). The IBM Personal Computer: First Impressions. *Byte*, 6(10), 26-34.
- Lockard, J., Abrams, P.D., & Many, W.A. (1987). *Microcomputers for educators*. Boston: Little, Brown and Company.
- London, R.L., & Duisberg, R.A. (1985). Animating programs using smalltalk. *IEEE Computer*, 18(8), 61-71.
- Long, K.T., & Squire, E.C. (1984). Courseware's Apple authoring system. *Journal of Computer-Based Instruction*, 11, 74-75.
- Lower, S.K. (1976). *Authoring languages and the evolution of C.A.I.*. Unpublished manuscript, Simon Fraser University, Department of Chemistry, Burnaby, BC.
- Mancinelli, B. (1987, July 23). The four dimensions of 4GL technology. *Computing Canada*, pp. 20, 24-25, 27.
- Marchionini, G. (1988). Special issue: Hypermedia. *Educational Technology*, 28(11).
- Martin, J. (1983-1984). *Fourth generation languages* (Vols. 1-2). Carnforth, Lancashire, England: Savant Research Studies.

- Martin, J., & McClure, C. (1984). *Diagramming techniques for analysts and programmers*. Englewood Cliffs, NJ: Prentice-Hall.
- Martin, M.P., & Fuerst, W.L. (1987). Using computer knowledge in the design of interactive systems. *International Journal of Man-Machine Studies*, 26, 333-342.
- Matsumura, K., & Tayama, S. (1986). Visual man-machine interface for program design and production. *1986 IEEE Computer Society Workshop on Visual Languages* (pp. 71-80). Washington: IEEE Computer Society Press.
- McMeen, G.R. (1986). Modern technology in education: From teaching machine to microcomputers and student response systems. *Educational Technology*, 26(8), 20-24.
- Merrill, M.D. (1985). Where is the authoring in authoring systems? *Journal of Computer-Based Instruction*, 12, 90-96.
- Merrill, M.D. (1987). Prescriptions for an authoring system. *Journal of Computer-Based Instruction*, 14, 1-10.
- Merrill, M.D., & Wood, L.E. (1984). Computer guided instructional design. *Journal of Computer-Based Instruction*, 11, 60-63.
- The Mitre Corporation. (1976). *An overview of the TICCIT program* (M76-44). Washington, DC: Author.
- Mizokawa, D.T., & Levin, J. (1988). Standards for error messages in educational software: An initial proposal. *Educational Technology*, 28(1), 19-24.
- Moriconi, M., & Hare, D.F. (1985). Visualizing program design through PegaSys. *IEEE Computer*, 18(8), 72-85.
- Mudrick, D., & Stone, D. (1984). An adaptive authoring system for computer-based instruction. *Journal of Computer-Based Instruction*, 11, 82-84.
- Murphy, R.T., & Appel, L.R. (1977). *Evaluation of the PLATO IV computer-based education system in the community college: Final report*. Washington: National Science Foundation.
- Myers, B.A. (1988). *Creating user interfaces by demonstration*. San Diego, CA: Academic Press.
- Niemiec, R.P., & Walberg, H.J. (1989). From teaching machines to microcomputers: Some milestones in the history of computer-based instruction. *Journal of Research on Computing in Education*, 21, 263-276.
- Norman, K.L., Weldon, L.J., & Shneiderman, B. (1986). Cognitive layouts of windows and multiple screens for user interfaces. *International Journal of Man-Machine Studies*, 25, 229-248.

- Olsen, D.R., Jr. (1986). Editing templates: A user interface generation tool. *IEEE Computer Graphics and Applications*, 6(11), 40-45.
- Paloian, A.Y. (1971). *An interrogative authoring system*. Unpublished master's thesis, University of Alberta, Edmonton.
- Park, O. (1988). Functional characteristics of intelligent computer-assisted instruction: Intelligent features. *Educational Technology*, 28(6), 7-14.
- Parkinson, S.R., Sisson, N., & Snowberry, K. (1985). Organization of broad computer menu displays. *International Journal of Man-Machine Studies*, 23, 689-697.
- Parnas, D.L., Clements, P.C., & Weiss, D.M. (1985). The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11, 259-266.
- Pogue, R.E. (1983). *Authoring systems and lesson transportability: The time has come*. Paper presented at the ADCIS Conference.
- Potosnak, K. (1987). Human factors: Creating software that people can and will use. *IEEE Software*, 4(5), 86-87.
- Pracht, W.E. (1986). GISMO: A visual problem-structuring and knowledge-organization tool. *IEEE Transactions on Systems, Man, and Cybernetics*, 16, 265-270.
- Pressman, I.S., & Pressman, D.E. (1986). Some advantages and disadvantages of MicroNATAL: Results from applications to the clinic environment. In *Proceedings of the Fifth Canadian Symposium on Instructional Technology* (pp. 348-354). Ottawa, ON: National Research Council Canada.
- Raeder, G. (1985). A survey of current graphical programming techniques. *IEEE Computer*, 18(8), 11-25.
- Randell, B. (1980). The Colossus. In N. Metropolis, J. Howlett, & G. Rota (Eds.), *A history of computing in the twentieth century: A collection of essays* (pp. 47-92). New York: Academic Press.
- Rath, G.J. (1967a). Computer teaching - 1967. *IEEE Transactions on Human Factors in Electronics*, 8, 59.
- Rath, G.J. (1967b). The development of computer-assisted instruction. *IEEE Transactions on Human Factors in Electronics*, 8, 60-63.
- Rath, G.J., Anderson, N.S., & Brainerd, R.C. (1959). The IBM research center teaching machine project. In E. Galanter (Ed.), *Automatic teaching: The state of the art* (pp. 117-130). New York: John Wiley & Sons.
- Reckase, M.D., & McKinley, R.L. (1982). *The validation of learning hierarchies* (Research Report ONR 82-2). Arlington, VA: Office of Naval Research, Personnel and Training Research Programs.
- Reed, M.J., & Porter, M. (1984). A review of Digital's courseware authoring system. *Journal of Computer-Based Instruction*, 11, 68-69.

- Reid, P. (1984). Work station design, activities and display techniques. In A. Monk (Ed.), *Fundamentals of human-computer interaction* (pp. 107-126). London: Academic Press.
- Reiss, S.P. (1985). PECAN: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, 11, 276-285.
- Romaniuk, E.W. (1970). *A versatile authoring language for teachers*. Unpublished doctoral dissertation, University of Alberta, Edmonton.
- Rushinek, A., & Rushinek, S.F. (1986). What makes users happy? *Communications of the ACM*, 29, 594-598.
- Sammet, J.E. (1986). Why Ada is not just another programming language. *Communications of the ACM*, 29, 722-732.
- Schneidewind, N.F. (1987). The state of software maintenance. *IEEE Transactions on Software Engineering*, 13, 303-310.
- Schwartz, J. (1983). Languages and systems for computer-aided instruction. *Machine-Mediated Learning*, 1, 5-39.
- Sherwood, B.A. (1977). *The TUTOR language*. Urbana, IL: University of Illinois, Computer-based Education Research Laboratory.
- Shirer, D.L. (1982). Evaluation of computer-based instruction systems. *Proceedings of the Conference of the Association for the Development of Computer-Based Instructional Systems* (pp. 5-11). Bellingham, WA: Western Washington University.
- Shneiderman, B. (1987). *Designing the user interface: Strategies for effective human-computer interaction*. Reading, MA: Addison-Wesley.
- Shu, N.C. (1988). *Visual programming*. New York: Van Nostrand Reinhold.
- Simpson, H. (1984). A human-factors style guide for program design. In D.F. Walker, & R.D. Hess (Eds.), *Instructional software: Principles and perspectives for design and use* (pp. 130-142). Belmont, CA: Wadsworth.
- Sleeman, D. (1985). UMFE: A user modelling front-end subsystem. *International Journal of Man-Machine Studies*, 23, 71-88.
- Stein, J., & Staiti, C. (Eds.). (1988). *CBT guide 1988: A directory of authoring systems and courseware*. Boston: Weingarten.
- Stern, R.H. (1989). Micro Law: Appropriate and inappropriate legal protection of user interfaces and screen displays, Part 1. *IEEE Micro*, 9(3), 84-88.
- Stetten, K.J., Morton, R.P., & Mayer, R.P. (1970). *The design and testing of a cost effective computer system for CAI/CMI application*. Washington: The Mitre Corporation.

- Suppes, P. (1984). Observations about the application of artificial intelligence research to education. In D.F. Walker, & R.D. Hess (Ed.), *Instructional software: Principles and perspectives for design and use* (pp. 298-308). Belmont, CA: Wadsworth.
- Tanimoto, S.L., & Glinert, E.P. (1986). Designing iconic programming systems: Representation and learnability. *1986 IEEE Computer Society Workshop on Visual Languages* (pp. 54-60). Washington: IEEE Computer Society Press.
- Technology news. (1986). *Educational Technology*, 26(8), 4.
- Teitelman, W. (1985). A tour through cedar. *IEEE Transactions on Software Engineering*, 11, 285-302.
- Thesen, A., & Beringer, D. (1986). Goodness-of-fit in the user-computer interface: A hierarchical control framework related to "friendliness". *IEEE Transactions on Systems, Man, and Cybernetics*, 16, 158-162.
- Thimbleby, H. (1984). User interface design: Generative user engineering principles. In A. Monk (Ed.), *Fundamentals of human-computer interaction* (pp. 165-180). London: Academic Press.
- Thompson, N. (1984a). Human memory: Different stores with different characteristics. In A. Monk (Ed.), *Fundamentals of human-computer interaction* (pp. 49-56). London: Academic Press.
- Thompson, N. (1984b). Thinking and reasoning: Why is logic so difficult? In A. Monk (Ed.), *Fundamentals of human-computer interaction* (pp. 57-63). London: Academic Press.
- Thompson, P. (1984). Visual perception: An intelligent system with limited bandwidth. In A. Monk (Ed.), *Fundamentals of human-computer interaction* (pp. 5-33). London: Academic Press.
- Verity, J.W. (1987). The OOPs revolution. *Datamation*, 33(5), 73-78.
- Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23, 459-494.
- Walker, D.F., & Hess, R.D. (Ed.). (1984). *Instructional software: Principles and perspectives for design and use*. Belmont, CA: Wadsworth.
- Wasserman, A.I., & Gutz, S. (1984). The future of programming. In D.F. Walker, & R.D. Hess (Ed.), *Instructional software: Principles and perspectives for design and use* (pp. 241-256). Belmont, CA: Wadsworth.
- Westrom, M.L. (1974). *National author language: NATAL-74: Author guide* (NRC 14243). Ottawa: National Research Council of Canada.
- Wexelblat, R.L. (Ed.). (1981). *History of programming languages*. New York: Academic Press.

- Wilson, B.G. (1985). Using content structure in course design. *Journal of Educational Technology Systems*, 14, 137-147.
- Yau, S.S., & Tsai, J.J.P. (1986). A survey of software design techniques. *IEEE Transactions on Software Engineering*, 12, 713-721.
- Yoshimoto, I., Monden, N., Hirakawa, M., Tanaka, M., & Ichikawa, T. (1986). Interactive iconic programming facility in HI-VISUAL. 1986 *IEEE Computer Society Workshop on Visual Languages* (pp. 34-41). Washington: IEEE Computer Society Press.
- Zinn, K.L. (1974). Requirements for effective authoring systems and assistance. *International Journal of Man-Machine Studies*, 6, 381-400.
- Zissos, A.Y., & Witten, I.H. (1985). User modelling for a computer coach: A case study. *International Journal of Man-Machine Studies*, 23, 729-750.

Appendix A Elf - The Educational Language Facility

developed at

Division of Educational Research Services

Faculty of Education

University of Alberta

Edmonton, CANADA

this description by

A.R. Davis and R.W.T. Garraway

The Educational Language Facility (Elf) is a systems programming environment intended for educators to use for the development of Computer-Assisted Learning systems. Such systems could also be applied to the field of Computer-Based Training in business and the military.

Elf System Components

The Elf programming environment consists of the following components:

- o The Elf language used to create interpreters. (These may be interpreters for existing computer languages, newly designed languages, application specific languages, or support systems.)
- o A data specification language used to define simple to complex data types.
- o A compiler for the Elf language. (This is currently a batch compiler but an interactive/visual incremental compiler is now under development.)
- o A machine independent systems environment in which Elf specified interpreters are used.

- o A machine dependent interface module, with complete debugging facilities, that "runs" the environment and the interpreters. (Currently, one has been developed for the Digital Equipment VAX and another is under development for the Macintosh II.)
- o A terminal independent multi-font and graphic window facility.
- o A window oriented text editor.

History of Elf Development

The Elf system has been under development by computer science and education specialists since November 1980. The first production system (version 4) came on line in July 1982. A higher performance production system (version 5) commenced operations in January 1984. When financial support for the DEC VAX facility in the Division of Educational Research Services was withdrawn in April 1990, development work continued on a Macintosh II. This version, 6, is an enhancement of the earlier work and is not yet in a production mode.

How Can Elf be Used?

It should be noted that Elf is NOT a CAI/CBT authoring language or authoring system. It is a systems programming environment that may be used to develop such languages and systems plus the necessary educational or training support facilities.

For example, under versions 4 and 5 of Elf, an enhanced Coursewriter II interpreter plus support facilities were written and implemented. (Coursewriter II was the CAI language used on the now decommissioned IBM 1500 Instructional System, one of which was used by the Division for CAI research, development, and production from 1968 to 1980.) Under version 5 an experimental authoring system has been developed that produces Coursewriter II code that is executed by the Coursewriter II interpreter mentioned above.

With the above system, a major credit course written in the IBM Coursewriter II language was offered at the University of Alberta via the DEC VAX and Gigi colour graphic terminals. This course consists of some 88000 lines of source code and some 474000 bytes of terminal graphics. It takes an average graduate student 90 hours to complete these CAI lessons.

Also, development work supported by a Social Sciences and Humanities Research Council grant (again using version 5 of Elf) for a new advanced visual and research oriented CAI system, ERAS (Educational Research Authoring System), was carried out.

Elf as a CAI/CBT Implementation Tool

Elf was developed to provide tools to implement the following model of Computer-Assisted Instruction and Computer-Based Training:

- o CAI/CBT courses consist of large data bases of text, pictures, sound, and instructional logic that must be conveyed to many students at video display (and, possibly, multi-media) terminals.
- o Student input is taken from the terminals, is analyzed, and in some fashion is used to control the presentation of the material in the data base.
- o A continuous stream of data about the events incoming from the terminals may be collected for later analysis.

Additional considerations are:

- o The same courseware may have to be delivered by different brands of hardware.
- o "Good" courseware will outlast the machine on which it was created and probably the author of that courseware.
- o A "course" is never finished.

Elf Design Goals

Some of the design goals for the Elf language and environment that have guided its development from the beginning of the project are:

1. The language should be easy to read, concise, and self documenting.
2. The learning time for new users of Elf should be short. It should not require a great deal of previous programming experience.
3. Debugging must be part of the language structure.
4. There should be a notation for the design of languages that forms a part of the implementation of those languages.

The Elf Language

Elf is a language to create interpreters for other languages. As such it contains constructs that are used for lexical analysis of input, constructs that define and manipulate data, perform arithmetic, etc., constructs that control the flow of execution, and mechanisms for relating to the outside world (i.e. the operating system on which it is running).

A program written in the Elf language is called a grammar. The form of an Elf grammar is a list of statements terminated by semi-colons and augmented by comments. The statements are the "productions" of an Elf grammar. All productions have a name. There are two types of productions. Productions named with an upper-case name are known externally to the grammar. These are called goal productions or the goals of the grammar. All other productions within a grammar, having lower-case names, are known internally and referred to as productions.

The productions provide a specification of input that is acceptable to a particular grammar. This requires an input stream called the "parse-buffer". The parse-buffer is a string that may contain one or more characters representing the

input of the language. (The language is defined by the Elf grammar accepting that input).

It takes the form of a grammar specifying the syntax of the language to be interpreted.

Elf Language Form

A specification of a language consists of several statements that specify the input syntax of the language that is to be interpreted and the actions that are to occur when various syntactic constructions occur. The form of these statements is based on Backus Naur Form (BNF) notation which is commonly used to specify the syntax of command languages, Pascal, Ada, Modula-2, etc.

A complete language specification will consist of several such statements each specifying in increasing detail the syntax of the target language. Each complete statement is called a "production" of the language.

All the productions that are used to describe a target language are called a "grammar".

Each production consists of a name followed by one or more parts called alternates and is terminated by a semi-colon.

A grammar must have at least one production distinct from all the others that is called a goal production. The goal production is known outside the grammar. It is not known inside the grammar. Goal productions are preceded by the term "goal" and take upper-case names, all other productions take lower-case names.

Each alternate in a production is made up of one or more entities called "terms". It is the terms that actually specify the details of the syntax and the semantics of the language.

Thus Elf productions have the following form:

<production_name>

```

is    term term term ... /* first alternate */
or    term term term ... /* second alternate */
:
:
or    term term term ... /* last alternate */ ;

```

(Comments are enclosed in "/*" and "*/".)

Terms are used to link the productions together in such a way as to make all productions in the grammar, starting at the goal production and working towards finer detail, a complete syntax specification.

The terms fall into several categories:

- o production_reference - used to link productions together inside a grammar.
In general, these take the form of a production_name enclosed in "<" and ">" (required syntactic element), "[" and "]" (optional syntactic element), or "[" and "]"..." (syntactic element that repeats 0 or more times).
- o grammar_reference - used to bring into a grammar an external syntax definition that is specified in another grammar. The general form of this type of term is: <grammar_name <GOAL_NAME>>
- o parse - used to direct analysis of input stream. Some examples of this type of term:
 - 0 or more of "abcdABCD"
 - 1 or more of "0123456789"
 - 1 to 12 of "!@#%\$^&*()"
 - 4 of "0123456789abcdefABCDEF"
 - 0 or more excluding "*/"

*integer

*float

- o semantic_actions - used to manipulate data, perform arithmetic, perform logical operations, etc. The general form is:

#semantic_name (result, arguments, ...)

- o control - used to modify execution of a grammar. Examples of control terms are: SUCCEED, FAIL_ALTERNATE, FAIL_PRODUCTION, PARSE_BUFFER (new_buffer_name).

To properly understand execution of a grammar, one views it as being entered at a goal production. The first alternate of the goal is examined term by term. Each term is executed in turn. A term must "succeed" or "fail".

When all the terms in an alternate succeed, the production that contains the alternate succeeds. If that is a goal production then the grammar is said to succeed. If any term fails then the alternate containing that term fails and the next alternate is executed. If all alternates in a production fail then the production fails. If that is a goal production then the grammar fails.

When a term is a reference term then it succeeds or fails depending on the result of executing the referenced production (or goal production).

The following Elf grammar is used to give a partial description of the syntax of an Elf grammar and the general flavour of the language.

```
goal <ELF_LANGUAGE>
  is  <comment>
  or  <body>;
```

```
<comment>
  is  "/*" 0 or more excluding ("*","/")
      <end_comment>;
```


All the syntactic descriptions in this thesis are written in the Elf language.

More technical information and demonstrations of both the Elf system and systems developed using Elf are available on request.

Appendix B ERAS Types, Variables, Constants, and Expressions

The syntax specifications in this section reflect the non-interactive declaration and listing forms of the ERAS language.

Types

Data are characterized by their type. Type defines both the range of values the data may take, their internal representation, and the way data may be referenced.

```
<type_declaration>
  is    "TYPES" <type_definition>
        [more_type_definitions]... ;
```

```
<more_type_definitions>
  is    ";" <type_definition> ;
```

```
<type_definition>
  is    <identifier> "=" <type> ;
```

```
<type>
  is    <simple_type>
  or    <structured_type> ;
```

```
<simple_type>
  is    <numeric_type>
  or    <string_type> ;
```

```
<numeric_type>
  is    <numeric_type_identifier>
  or    "NUMERIC" [initial_numeric_value] ;
```

```
<initial_numeric_value>
  is    ":@" <numeric_expression> ;
```

/* Implemented as the Elf INTEGER type but represents a fixed point decimal number with three decimal places. A numeric is stored as an integer equal to the numeric value time 1000. */

```
<string_type>
  is    <string_type_identifier>
  or    "STRING" "OF" <max_string_length>
        [initial_string_value] ;
```

```
<initial_string_value>
  is    ":@" <string_expression> ;
```



```

<max_string_length>
  is    <numeric_expression>
        /* range 1 to 32767 */;

<structured_type>
  is    <list_type>
  or    <record_type>;

<list_type>
  is    <list_type_identifier>
  or    "LIST" "[" <max_list_elements> "]" "OF"
        <type>;

<max_list_elements>
  is    <numeric_expression>;

<record_type>
  is    <record_type_identifier>
  or    "RECORD" <field>
        [more_fields]... "END_RECORD";

<more_fields>
  is    ";" <field>;

<field>
  is    <field_identifier>
        ":" <type>;

```

System Types

In addition to the primitive and structured types given above, the system may define other types. Below are some suggested predefined system types:

```

/* SYSTEM */ TYPES
  POINTER = RECORD
    R : NUMERIC
    C : NUMERIC
  END_RECORD;

  MENU = LIST [30] OF POINTER;

  MENU_SET = LIST [30] OF MENU

```

Variables

```

<variable_declaration>
  is    "VARIABLES" <variable_definition>
        [more_variable_definitions]...;

```



```
<more_variable_definitions>
  is      ";" <variable_definition>;
```

```
<variable_definition>
  is      <variable_identifier>
          ":" <type>;
```

Dynamic Variables

Variables of POINTER type may have space allocated to them dynamically at execution time by a POINT statement.

```
<point_statement>
  is      "POINT" <pointer_variable>
          "TO" <qualified_data_element>;
```

```
<pointer_variable>
  is      <identifier>;
```

Constants

```
<constant_declaration>
  is      "CONSTANTS" <constant_definition>
          [more_constant_definitions]...;
```

```
<more_constant_definitions>
  is      ";" <constant_definition>;
```

```
<constant_definition>
  is      <identifier> "=" <constant_value>;
```

```
<constant_value>
  is      [sign] <numeric_value>
  or      <boolean_constant>
  or      <string_value>
  or      <expression>;
```

```
<sign>
  is      "+"
  or      "-";
```

```
<numeric_value>
  is      *integer /* <= 2147482 */
          [fraction_part];
```

```
<fraction_part>
  is      "." *integer /* <= 999 */;
```



```

<boolean_constant>
  is  "TRUE"
  or  "FALSE"
  or  "YES"
  or  "NO"
  or  "ON"
  or  "OFF" ;

```

```

<string_value>
  is  "" 0 or more excluding "" <end_string> ;

```

```

<end_string>
  is  "" ""
      0 or more excluding "" <end_string>
  or  "" ;

```

/* A string_value is enclosed by a quote character on each end that is not part of the string. Two consecutive quote characters within a string_value represent one quote character and take only one position in the string. */

System Constants

There will be a number of system constants pre-declared. A partial list follows:

MAXINT -	the largest positive integer supported by the system (2147482)
MININT -	the lowest negative integer support by the system (-2147482)
INFINITY -	the largest unsigned real number supported by the system (2147482.999)
EPSILON -	the smallest unsigned fractional real number supported by the system; a smaller number would be represented by 0 (0.001)

Assignment and Expressions

This section briefly outlines the facilities available in assignment statements and expressions of the ERAS language. In the following discussion, simplified syntactic specifications will be used. Items in lower case are syntactic element descriptors. Items in upper case are actual symbols to be used. Syntactic elements may be grouped by enclosing them in braces, { ... }, indicating that they are required; or in

brackets, [...], indicating that they are optional. If the right brace is followed by an ellipsis, }..., then the enclosed syntactic elements may be repeated but must appear at least once. If the right bracket is followed by an ellipsis,]..., then the enclosed syntactic elements may be repeated zero or more times. If a brace or a bracket is a syntactic element it will be enclosed in quotation marks, i.e. "{", "}", "[", "]". The vertical bar, |, separates groups of alternate syntactic elements. All other symbols should be considered syntactic elements.

The Assignment Statement

The basic form is:

```
qualified_data_element assignment_operator expression
```

Numeric assignment operators are:

```
:=    assign numeric value
:+    increment numeric value
:-    decrement numeric value
:C=   convert numeric in string to numeric value
```

String assignment operators are:

```
:=    assign string value
:+    append with no intervening space
:&    append with one intervening space
:C=   convert numeric value to its string representation
```


The assignment operators were chosen because they are similar to those used in structured languages, e.i., ALGOL, COMAL, Modula 2, and Pascal.

String Expressions

The form of a string expression is:

```
string_term [ string_operator string_term ]...
```

The form of a string term is:

```
string_factor [ string_multiplier ]
```

The form of a string multiplier is:

```
* numeric_expression
```

The effect of the optional string multiplier is to concatenate a number of copies of the string factor, e.g.,

```
"fred" * 3 is "fredfredfred"
```

The string operators are:

```
+ concatenate with no intervening space
& concatenate with one intervening space
```


The form of a string factor is:

```
string_value
string_constant_identifier
qualified_data_element
string_function
( string_expression )
```

A string value is 0 or more printable characters enclosed in double quotation marks. The quotation mark is itself represented by two successive quotation marks. For example, the string assignment:

```
str := "John called out, ""Hi Mom!"""
```

would assign the following string literal to variable 'str':

```
John called out, "Hi Mom!"
```

The string qualified data element and the string function call may be followed by an optional subpart specification. The form of the subpart specification is:

```
"{ " [ left_offset ] [ ^ right_offset | ~ length ] " }
```

Left offset, right offset, and length are numeric expressions. A special system variable, \$, representing the current length of the string being subparted, may participate in these numeric expressions. The left offset defaults to 1 and the right part defaults to ^\$. The index origin of a character string 1.

In the definition of a numeric expression that follows, at the relational operator precedence level a string relation may be substituted for a numeric relation. That is, the following two statements are equivalent as far as their precedence level participation in a numeric expression is concerned:

```
arithmetic_exp relational_operator arithmetic_exp
string_exp string_relational_operator string_exp
```

There is an additional string relational operator besides the normal relational operators. The form is as follows:

```
sting_exp_1 IN string_exp_2
```

This relation returns the position of sting_exp_1 in string_exp_2. If string_exp_1 is not in string_exp_2, it returns 0. (0 is FALSE and anything other than 0 is TRUE.)

Numeric Expressions

An expression that results in a numeric value is a numeric expression. Boolean expressions are considered as numeric expressions because boolean values are automatically converted to 1 (TRUE) and 0 (FALSE).

The general form of a numeric expression is as follows:

```
operand [ operator operand ]...
```

The special exception is the participation of a string relation at the relational operator precedence level. The order of precedence, from highest to lowest, of numeric operators is as follows:

1. unary + or -
2. * / DIV MOD
3. + -
4. = < > <= >=
5. unary NOT ~
6. AND &
7. OR XOR |

DIV is integer division and MOD is remainder after integer division.

The form of a numeric operand is as follows:

numeric_value

boolean_value

numeric_constant_identifier

qualified_data_element

numeric_function

(numeric_expression)

The following are equivalent boolean values that equal 1.000:

TRUE YES ON 1

The following are equivalent boolean values that equal 0:

FALSE NO OFF 0

The form for a numeric value is:

[sign] whole_number_part [. fractional_part]

(NOTE: no spaces allowed within number)

sign is + or -

whole_number_part between 0 and 2147482, inc.

fractional_part between 0 and 999, inc.

All numerics are stored as integers equal to the numeric times 1000.

Numeric Conversion Rules

In an integer expression, a boolean value is converted to 1 (TRUE) or 0 (FALSE). In a real expression, an integer or boolean value is converted to a real value. In a boolean expression, a numeric value of 0 is converted to FALSE and any other numeric value is converted to TRUE.

Qualified Data Element

The form of a qualified data element is as follows:

variable_identifier [list_index]... [field_reference]...

The form of a field reference is:

.field_identifier [list_index]...

(NOTE: the *dot* must exactly precede the field identifier)

The form of a list index is:

"[" location [, location]... "]"

A location may be a numeric expression or one of the following key words:

FIRST LAST NEXT PREV EOL BOL EMPTY

The first four are used to obtain the referenced element in the list. NEXT and PREV have no meaning until the list has been entered by FIRST or LAST. If no such element exists, an error is generated. The last three return a boolean result for the referenced list, i.e.

EOL - is the list pointer at the End Of the List?

BOL - is the list pointer at the Beginning Of the List?

EMPTY - is the list empty?

Appendix C ERAS Control Language

Introduction

The control language is used to define control modules, sub-routines (usually just called routines), and functions. It is specified as a structured procedure oriented language modeled on Pascal and COMAL (COMMon Algorithmic Language). There is one control module at each scope level in a course and it is used to direct the flow of the student through the content modules of that level plus the invocation of control modules at the next lower level. Entry into a course is through the course control module. Routines and functions allow for the definition of algorithmic procedures. Routines are invoked by a ROUTINE statement. Functions return a STRING or NUMERIC explicit result and are call from within <expression>'s.

Syntax

The syntax specifications below represent the non-interactive declarations and listing forms of the language.

```
<process>
  is    <control_process>
  or    <routine_process>
  or    <function_process> ;

<control_process>
  is    "CONTROL" <identifier> <eol>
        [parameter]...
        [local_constant]...
        [local_variable]...
        [statement]...
        "END_CONTROL" <identifier> <eol> ;

<routine_process>
  is    "ROUTINE" <identifier> <eol>
        [parameter]...
        [local_constant]...
        [local_variable]...
        [statement]...
        "END_ROUTINE" <identifier> <eol> ;
```



```

<function_process>
  is    "FUNCTION" <identifier> ":" <function_type>
        <eol>
        [parameter]...
        [local_constant]...
        [local_variable]...
        [statement]...
        "END_FUNCTION" <identifier> <eol> ;

```

```

<function_type>
  is    "NUMERIC"
  or    "STRING" ;

```

```

<parameter>
  is    <value_parameter>
  or    <reference_parameter> ;

```

```

<value_parameter>
  is    "VAL" <identifier> ":"
        <value_type> [default_value] <eol> ;

```

```

<value_type>
  is    "NUMERIC"
  or    "STRING" "OF" <numeric_expression> ;

```

```

<default_value>
  is    "!=" <expression> ;

```

```

<reference_parameter>
  is    "REF" <identifier> ":"
        <reference_type> <eol> ;

```

```

<reference_type>
  is    "NUMERIC" [default_value_reference]
  or    "STRING" [default_value_reference]
  or    "LIST" [default_reference]
  or    "RECORD" [default_reference]
  or    <type_identifier> [default_reference] ;

```

```

<default_value_reference>
  is    <default_value>
  or    <default_reference> ;

```

```

<default_reference>
  is    "\" <qualified_data_element> ;

```

```

<type_identifier>
  is    <identifier> ;

```


Process Body

```
<local_constant>
  is    "CON" <identifier> "="
        <value> <eol> ;
```

```
<local_variable>
  is    "VAR" <identifier> ":"
        <type> <eol> ;
```

```
<type>
  is    /* See Appendix B */;
```

```
<statement>
  is    <structure_statement>
  or    <simple_statement> <eol> ;
```

```
<simple_statement>
  is    <comment>
  or    <null>
  or    <content_call>
  or    <routine_call>
  or    <system_routine_call>
  or    <control_call> /* only in CONTROL modules */
  or    <loop_exit>
  or    <return_from_process>
  or    <pointer>
  or    <assignment> ;
```

```
<structure_statement>
  is    <loop_structure>
  or    <while_structure>
  or    <until_structure>
  or    <for_structure>
  or    <if_structure>
  or    <case_structure> ;
```

Simple Statements

```
<comment>
  is    "///"
        0 or more of anything ;
```

```
<null>
  is    "NULL" ;
```

```
<loop_exit>
  is    "EXIT" "WHEN"
        <numeric_expression>
        /* conditional loop exit */
```



```

    or      "EXIT"
           /* unconditional loop exit */;

<return_from_process>
    is      "RETURN" [expression] ;

<pointer>
    is      "POINT" <identifier>
           "TO" <qualified_data_element> ;

```

Process Calls

```

<content_call>

/* This will be a call to the content module interpreter */

    is      "CONTENT" <identifier>
           [actual_parameters] ;

<routine_call>
    is      "ROUTINE" <identifier>
           [actual_parameters] ;

<system_routine_call>
    is      "&"<identifier>
           [actual_parameters] ;

<control_call>
    is      "CONTROL" <identifier>
           [actual_parameters] ;

<actual_parameters>
    is      "(" <actual_parameter>
           [more_parameters]... ")" ;

<more_parameters>
    is      "," <actual_parameter> ;

<actual_parameter>

/* If an actual parameter is missing, the formal parameter is checked for a predefined
default. If no default is defined, an error will occur. */

    is      <actual_value_parameter>
    or      <actual_reference_parameter> ;

<actual_value_parameter>
    is      <expression> ;

<actual_reference_parameter>
    is      <qualified_data_element> ;

```


Assignment Statement

```

<assignment>
  is    <qualified_data_element>
        <assign_operator>
        <expression> ;

<assign_operator>
  is    "：=" /* NUMERIC or STRING assignment */
  or    "：+" /* NUMERIC increment and STRING append */
  or    "：-" /* NUMERIC decrement */
  or    "：&" /* STRING append with intervening space */
  or    "：C=" /* With type conversion */ ;

```

Structure Statements

```

<loop_structure>
  is    "LOOP" <eol>
        <statement>
        [statement]...
        "ENDLOOP" <eol> ;

<while_structure>
  is    "WHILE" <numeric_expression>
        "DO" <eol>
        <statement>
        [statement]...
        "ENDWHILE" <eol> ;

<until_structure>
  is    "REPEAT" <eol>
        <statement>
        [statement]...
        "UNTIL" <numeric_expression> <eol> ;

<for_structure>
  is    "FOR" <qualified_data_element>
        "：=" <numeric_expression>
        "TO" <numeric_expression>
        [step]
        "DO" <eol>
        <statement>
        [statement]...
        "ENDFOR" <qualified_data_element> <eol> ;

<step>
  is    "STEP" <numeric_expression> ;

```



```

<if_structure>
    is    "IF" <numeric_expression>
          "THEN" <eol>
          <statement>
          [statement]...
          [elif_structure]...
          [else_structure]
          "ENDIF" <eol> ;

<elif_structure>
    is    "ELIF" <numeric_expression>
          "THEN" <eol>
          <statement>
          [statement]... ;

<else_structure>
    is    "ELSE" <eol>
          <statement>
          [statement]... ;

<case_structure>
    is    "CASE" <qualified_data_element> <eol>
          [when_structure]...
          [otherwise_structure]
          "ENDCASE" <qualified_data_element> <eol> ;

<when_structure>
    is    "WHEN" <expression>
          [more_expressions]...
          "DO" <eol>
          <statement>
          [statement]... ;

<more_expressions>
    is    "," <expression> ;

<otherwise_structure>
    is    "OTHERWISE" <eol>
          <statement>
          [statement]... ;

<eol>
    is    [comment] *eol ;

```

Expression Evaluator

```

<expression>
    is    <string_expression>
    or    <numeric_expression> ;

```



```

<value>
  is   <string_value>
  or   <numeric_value>
  or   <boolean_value>;

<string_value>
  is   "" 0 or more excluding "" <end_string>;

<end_string>
  is   "" ""
      0 or more excluding "" <end_string>
  or   "" ;
      /* the quote character is represented
         by "" in a string */

<numeric_value>
  is   [sign]
      *integer /* <= 2147482 */
      [fraction_part];

<sign>
  is   "+"
  or   "-";

<fraction_part>
  is   "." *integer /* <= 999 */;

<boolean_value>
  is   "TRUE"
  or   "FALSE"
  or   "YES"
  or   "NO"
  or   "ON"
  or   "OFF";

```

String Expression Evaluator

```

<string_expression>
  is   <string_term>
      [more_string_terms]...;

<more_string_terms>
  is   "+" <string_term> /* concatenate */
  or   "&" <string_term> /* concatenate with space */;

<string_term>
  is   <string_factor>
      [string_multiplier];

<string_multiplier>
  is   "*" <numeric_expression>;

```



```

<string_factor>
  is  "(" <string_expression> ")"
  or  <string_value>
  or  <string_constant_identifier>
  or  "!"<string_function_call>
  or  "&"<system_string_function_call>
  or  <qualified_data_element>
      [subpart_specification] ;

```

```

<string_function_call>
  is  <identifier>
      [actual_parameters]
      [subpart_specification] ;

```

```

<system_string_function_call>
  is  <identifier>
      [actual_parameters]
      [subpart_specification] ;

```

```

<subpart_specification>
  is  "{" [left_offset]
      [right_part] "}"
      /* $ may be used in any subpart
         numeric expression; it is a special
         variable that represents the current
         length of the string */;

```

```

<left_offset>
  is  <numeric_expression>
      /* defaults to 1 */;

```

```

<right_part>
  is  "^" <right_offset>
  or  "~" <length>
      /* defaults to ^$ */;

```

```

<right_offset>
  is  <numeric_expression> ;

```

```

<length>
  is  <numeric_expression> ;

```

Numeric Expression Evaluator

```

<numeric_expression>
  is  <logical_term>
      [more_logical_terms]... ;

```



```

<more_logical_terms>
  is  "OR"@ <logical_term>
  or  "|" <logical_term>
  or  "XOR"@ <logical_term> ;

<logical_term>
  is  <logical_factor>
      [more_logical_factors]... ;

<more_logical_factors>
  is  "AND"@ <logical_factor>
  or  "&" <logical_factor> ;

<logical_factor>
  is  [negate] <relation> ;

<negate>
  is  "NOT"@
  or  "~" ;

<relation>
  is  <arithmetic_relation>
  or  <string_relation> ;

<string_relation>
  is  <string_expression>
      <string_relational_operator>
      <string_expression> ;

<string_relational_operator>
  is  "IN"
  or  <relational_operator> ;

<arithmetic_relation>
  is  <formula>
      [formula_relation] ;

<formula_relation>
  <relational_operator>
  <formula> ;

<relational_operator>
  is  "<"
  or  "<="
  or  "<"
  or  ">="
  or  ">"
  or  "=" ;

<formula>
  is  [sign] <arithmetic_expression> ;

<arithmetic_expression>
  is  <term> [more_terms]... ;

```



```

<more_terms>
    is    <sign> <term> ;

<sign>
    is    "+"
    or    "-";

<term>
    is    <factor> [more_factors]... ;

<more_factors>
    is    <multiplicative_operator>
          <factor> ;

<multiplicative_operator>
    is    "*"
    or    "/"
    or    "DIV"
    or    "MOD" ;

<factor>
    is    "(" <numeric_expression> ")"
    or    <boolean_value>
    or    <numeric_value>
    or    <constant_identifier>
    or    "!"<numeric_function_call>
    or    "&"<system_numeric_function_call>
    or    <qualified_date_element> ;

<numeric_function_call>
    is    <identifier>
          [actual_parameters] ;

<system_numeric_function_call>
    is    <identifier>
          [actual_parameters] ;

```

Qualified Data Element Evaluator

```

<qualified_data_element>
    is    <variable_name> [list_index]...
          [field_reference]... ;

<field_reference>
    is    "." <field_identifier>
          [list_index]... ;

<field_identifier>
    is    <identifier> ;

```


<list_index>
is "[" <numeric_expression>
 [more_list_index]... "]" ;

<more_list_index>
is ", " <numeric_expression> ;

<variable_name>
is <identifier> ;

<identifier>
is 1 of ("abcdefghijklmnopqrstuvwxyz"
 "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
 "_#\$%")
 0 to 22 of ("abcdefghijklmnopqrstuvwxyz"
 "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
 "0123456789_#\$%") ;

Appendix D ERAS Content Language

Introduction

The content language is provided for the structuring of content modules. The presentation of information to the student and the processing of student responses to questions is handled by content modules. A content module presents the content of a small portion of a course in a linear sequence selected by the author. The elements in this sequence may be displays of text and graphics, the posing of questions, and the analysis, classification, and reprise to the student's response. Intermediate computations and the calling of subroutines may also be done within a content module. Content modules may be invoked by other content modules, and by control and routine modules which control the general flow of a student through a course.

Syntax

The syntax specifications below represent the non-interactive declarations and listing forms of the language.

```
<content_process>
  is  "CONTENT:" <identifier> <eol>
      [parameter]...
      [local_constant]...
      [local_variable]...
      [statement]...
      "END_CONTENT" <identifier> <eol> ;
```

```
<parameter>
  is  <value_parameter>
  or  <reference_parameter> ;
```

```
<value_parameter>
  is  "VAL" <identifier> ":"
      <value_type> <eol> ;
```

```
<value_type>
  is  "NUMERIC"
  or  "STRING" "OF" <numeric_expression> ;
```



```

<reference_parameter>
  is    "REF" <identifier> ":"
        <reference_type> <eol> ;

```

```

<reference_type>
  is    "NUMERIC"
  or    "STRING"
  or    "LIST"
  or    "RECORD"
  or    <type_identifier> ;

```

```

<type_identifier>
  is    <identifier> ;

```

Process Body

```

<local_constant>
  is    "CON" <identifier> "="
        <value> <eol> ;

```

```

<value>
  is    /* See Appendix C */ ;

```

```

<local_variable>
  is    "VAR" <identifier> ":"
        <type> <eol> ;

```

```

<type>
  is    /* See Appendix B */ ;

```

```

<statement>
  is    <simple_statement> <eol> ;

```

```

<eol>
  is    [comment] *eol ;

```

```

<simple_statement>
  is    <comment>
  or    <pointer>
  or    <assignment>
  or    <routine_call>
  or    <content_call>
  or    <display_call>
  or    <input_call>
  or    <answer_call>
  or    <menu_call> ;

```


Simple Statements

```

<comment>
    is    "///"
           0 or more of anything ;

<pointer>
    is    "POINT" <identifier>
           "TO" <qualified_data_element> ;

<assignment>
    is    <qualified_data_element>
           <assign_operator>
           <expression> ;

<assign_operator>
    is    /* See Appendix C */ ;

<expression>
    is    /* See Appendix C */ ;

<qualified_data_element>
    is    /* See Appendix C */ ;

<identifier>
    is    1 to 32 of      ("abcdefghijklmnopqrstuvwxyz"
                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                           "0123456789_#$%");

```

Process Calls

```

<routine_call>
/* This is a call to the control language interpreter */

    is    "ROUTINE" <identifier>
           [actual_parameters] ;

<content_call>
/* This is a call to the content language interpreter */

    is    "CONTENT" <identifier>
           [actual_parameters] ;

<display_call>
/* This is a call to the display language interpreter */

    is    "DISPLAY"
           <display_body> ;

```



```

<display_body>
  is    <identifier>
        [actual_parameters]
  or    <eol>
        [display_statement]...
        "END_DISPLAY" ;

<display_statement>
  is    /* from the DISPLAY Language */ ;

<input_call>
/* This is a call to the input language interpreter */

  is    "INPUT"
        <input_body> ;

<input_body>
  is    <identifier>
        [actual_parameters]
  or    <eol>
        [input_statement]...
        "END_INPUT" ;

<input_statement>
  is    /* from the INPUT Language */ ;

<answer_call>
/* This is a call to the answer language interpreter */

  is    "ANSWER"
        <answer_body> ;

<answer_body>
  is    <identifier>
        [actual_parameters]
  or    <eol>
        [answer_statement]...
        "END_ANSWER" ;

<answer_statement>
  is    /* See Appendix E */ ;

<menu_call>
/* This is a call to the menu language interpreter */

  is    "MENU"
        <menu_body> ;

<menu_body>
  is    <identifier>
        [actual_parameters]
  or    <eol>
        [menu_statement]...
        "END_MENU" ;

```


<menu_statement>
is /* from the MENU Language */;

<actual_parameters>
is "(" <actual_parameter>
[more_parameters]... ")" ;

<more_parameters>
is "," <actual_parameter> ;

<actual_parameter>
is <actual_value_parameter>
or <actual_reference_parameter> ;

<actual_value_parameter>
is <expression> ;

<actual_reference_parameter>
is <qualified_data_element> ;

Appendix E ERAS Answer Language

Introduction

The answer language is provided for the structuring of answer modules. The answer language provides the means to accept a student response, to classify the response for judging, and to provide feedback to the student based on the classified response.

Answer statements may also appear as lines between an ANSWER line and an END_ANSWER line in a CONTENT module.

Syntax

An ANSWER module has the following features:

- o There must be exactly one INPUT statement at the outer level of every ANSWER
- o The rest of the ANSWER is composed of any number of simple_statements (see Appendix D) and answer condition statements called acons.
- o An acon is a specialized control structure unique to the ANSWER language.

The syntax specifications below represent the non-interactive declarations and listing forms of the language.

```
<answer_process>
  is  "ANSWER:" <identifier> <eol>
      [parameter]...
      [local_constant]...
      [local_variable]...
      [answer_statement]...
      <input_statement> /* from INPUT Language */
      [answer_statement]...
      "END_ANSWER" <identifier> <eol>
```



```

<parameter>
  is    <value_parameter>
  or    <reference_parameter> ;

<value_parameter>
  is    "VAL" <identifier> ":"
        <value_type> <eol> ;

<value_type>
  is    "NUMERIC"
  or    "STRING" "OF" <numeric_expression> ;

<reference_parameter>
  is    "REF" <identifier> ":"
        <reference_type> <eol> ;

<reference_type>
  is    "NUMERIC"
  or    "STRING"
  or    "LIST"
  or    "RECORD"
  or    <type_identifier> ;

<type_identifier>
  is    <identifier> ;

<local_constant>
  is    "CON" <identifier> "="
        <value> <eol> ;

<value>
  is    /* See Appendix C */ ;

<local_variable>
  is    "VAR" <identifier> ":"
        <type> <eol> ;

<type>
  is    /* See Appendix B */ ;

<answer_statement>
  is    <acon>
  or    <simple_statement> /* see Appendix D */ ;

<acon>
  is    <category_name> <numeric_expression> <eol>
        <acon_action_part> ;

<category_name>
  is    <identifier> ;

<acon_action_part>
  is    [acon_action]... ;

```



```

<acon_action>
  is    "DO" <eol>
        [simple_statement]...
        <acon_branch> ;

<acon_branch>
  is    "RETRY" /* goto INPUT reference */
  or    "EXIT"  /* exit ANSWER          */
  or    "CONT"  /* goto next statement */;

```

Categories

A category name identifies a variable of type CATEGORY.

CATEGORY is a predefined record type consisting of:

STATUS - an integer set to 0 at entry to current scope

COUNTER - an integer set to 0 at entry to current scope

When the answer condition statement is executed the STATUS is set to the value of the numeric_expression. If the expression is true the COUNTER is incremented by 1 and the action_part is executed. If the expression is not true the COUNTER is not incremented, the action_part is not executed, and the acon passes control to the following statement.

There is only one predefined category. Its name is IF.

Acon Action

Every acon has its own pointer which points to the next action to be executed. The pointer is initialized to the first action at entry to the scope from which the acon is referenced. When an action_part is executed the pointer is updated to point to the next action. When all the actions have been used up the pointer remains at the last action.

Example:

```

ANSWER: quest3

REF num_right : NUMERIC
VAR OK : CATEGORY
VAR UNREC : CATEGORY

INPUT ("What is the capital of China?")

OK &scan("Beijing") OR &scan("Peking")
  DO
    DISPLAY right3
    num_right :+1
  EXIT

UNREC TRUE
  DO
    DISPLAY unrec1
  RETRY

  DO
    DISPLAY unrec2
  RETRY

  DO
    DISPLAY unrec3
  RETRY

END_ANSWER quest3

```

If there is a match on the 'OK' category, the acon_action (DISPLAY right3, num_right :+1) is carried out and then ANSWER module 'quest3' is EXITed. If there is no match on 'OK' there will be a forced match on category 'UNREC'. On the first 'UNREC' match the acon_action (DISPLAY unrec1) is carried out and there is a branch back to the input statement. On the second 'UNREC' match the acon_action (DISPLAY unrec2) is carried out and there is a branch back to the input statement. On all subsequent 'UNREC' matches the acon_action (DISPLAY unrec3) is carried out followed by a branch back to the input statement.

JUN 8 1993

LB 1028-66 G23 1993
GARRAWAY ROBERT WILLIAM THOMAS
1938-

HIERARCHICAL CONTROL AND
M1 40212833 EDUC



000041973322

DATE DUE SLIP

APR 10 '97

AUG 25 '97

RETURN NOV 26 '98

RETURN APR 10 '97

RETURN APR 10 '97

RETURN APR 20 '98

RETURN APR 20 '98

